ESD-TR-77-258

MTR-3269

TOP LEVEL SPECIFICATION OF A SECURITY KERNEL FOR MULTICS FRONT-END PROCESSOR

NOVEMBER 1977

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Project No. 522N

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. AF19628-77-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

William R Price
WILLIAM R. PRICE, Capt, USAF
Techniques Engineering Division

[illegible]
ROGER R. SCHELL, Lt Col, USAF
ADP System Security Program Manager

FOR THE COMMANDER

[illegible]
STANLEY P. DERESKA, Colonel, USAF
Deputy Director, Computer Systems Engineering
Deputy for Command and Management Systems

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-77-258	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* TOP LEVEL SPECIFICATION OF A SECURITY KERNEL FOR MULTICS FRONT-END PROCESSOR		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER MTR-3269
7. AUTHOR*(s)* M. Gasser		8. CONTRACT OR GRANT NUMBER*(s)* AF19628-77-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Box 208 Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 522N
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division (AFSC) Hanscom Air Force Base, MA 01731		12. REPORT DATE NOVEMBER 1977 13. NUMBER OF PAGES 110
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

FRONT-END PROCESSOR
MULTICS
OPERATING SYSTEMS
SECURITY

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A security kernel is a combination of hardware and software that controls access to information within a computer system. The top level specification is a formal description of the interfaces between the security kernel and nonkernel software, and used in providing the correctness of the kernel with respect to security requirements. The specification ultimately becomes the criterion against which the correctness of

20. ABSTRACT (concluded)

the implementation is verified. This report gives the top level specifications of a security kernel for the front-end processor for a secure Multics system. The hardware configuration that will support secure Multics is a Honeywell Series 60 (Level 68) mainframe with a Level 6 minicomputer as the front-end processor. A separate report deals with the Multics security kernel.

ACKNOWLEDGMENTS

This report has been prepared by The MITRE Corporation under Project No. 522N. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts

TABLE OF CONTENTS

TABLE OF CONTENTS (concluded)

LIST OF ILLUSTRATIONS

SECTION I

INTRODUCTION

Honeywell's Multics System is currently undergoing redesign that will allow it to simultaneously process information of different classifications, a mode of operation known as multi-level processing. In order for a computer system to be certified for multi-level operation, it must be proven that classified information in the system cannot be compromised. The key to development of a certifiable system is the concept of a reference monitor, embodied in a security kernel [1]. The kernel is a protected portion of the operating system and hardware that is small enough so that it can be mathematically verified to operate correctly and to provide the required security controls.

A top level specification of a security kernel is an abstract description of the interfaces between the kernel and user software. The specification is used as the basis for mathematical verification. This report provides the top level specification of a security kernel for a secure front-end processor (SFEP) for Multics. The top level specification for the Multics security kernel is currently in preparation.

Although extensive knowledge of Multics is not required for an understanding of this report, it is assumed that the reader is familiar with the basic concepts of Multics and related terminology [2,3].

BACKGROUND

The first operational security kernel has been implemented on a PDP-11/45 minicomputer [4] to demonstrate the feasibility of applying the kernel concept in a real operating system. Although the operating system utilizing the kernel is primitive, the kernel itself is general enough to show that it is indeed possible to construct a reasonably small kernel that could support a relatively complex operating system.

While the PDP-11/45 kernel was being developed, the Air Force, MITRE, and Honeywell collaborated on the design for a set of "security enhancements" that could be incorporated into Multics to simulate as closely as possible the envisioned operation of a kernel based Multics [5]. These security enhancements are collectively referred to as the Access Isolation Mechanism, and have become part of the Multics standard product [3]. Though not formally specified or validated, the enhancements were carefully designed to provide the highest degree of security possible within the existing Multics framework.

The purpose of designing enhancements for this interim system was to gain insight into the problems that might be encountered by users who have to work with a multi-level system, and to fulfill an immediate need for a multi-level operational capability at the Air Force Data Services Center. Multics with the Access Isolation Mechanism will soon be authorized for multi-level processing at the AFDSC in a limited environment that minimizes the threat of malicious attack by restricting access to the system to secret and top secret cleared individuals.

Multics is basically a remotely accessed system and has a front-end processor that handles the communications interfaces, mostly to terminals. Currently the Series 60 (Level 68) Multics processor uses a DataNet 6600 as its front-end. Since the front-end processor is itself a computer, its operation must also be based on a security kernel if it is to be involved in multi-level processing. Unfortunately, the DataNet 6600 cannot support a kernel because the hardware lacks various features seen essential to the efficient implementation of a kernel [6]. Instead, the decision was made to provide the necessary hardware support by designing a "security protection module" (SPM) that could be logically added onto an existing minicomputer to provide the necessary hardware features. The Honeywell Series 60 (Level 6) minicomputer family [7] has been chosen as the hardware base to be enhanced by the addition of an SPM.

KERNEL DESIGN PROCESS

The construction of a security kernel begins with a mathematical model that provides the axioms and properties of a system that implements a specific security policy. Such a model has been developed for the Department of Defense security regulations [8] for a computer system with a Multics-like structure.

Using the model as a guide, an abstract specification of the kernel's interfaces with non-kernel software and users is prepared. This interface is the top level specification. From this top level, additional details of the kernel's design, embodied in lower level specifications, are prepared, and finally the computer programs are written to implement the specification.

In conjunction with the kernel design and implementation, the validation process provides the proof that the design and implementation are secure. The proof consists of several phases: proving correspondence of the top level specification to the model, proving that the kernel satisfies certain properties of a reference monitor (i.e., that it mediates all accesses and is isolated), and that the implementation of the kernel corresponds to the specification.

In order to be able to validate any system, it must be as small and simple as possible. For this reason a great deal of effort goes into minimizing the size and complexity of the kernel specification. It is hoped that the end result will be a verifiable kernel that contains the minimal mechanism necessary that efficiently supports an operating system and provides the required controls.

SECTION II

GENERAL CONCEPTS AND DEFINITIONS

As a prerequisite for understanding of the specification, the reader must be familiar with certain underlying concepts of kernel design. These concepts have been discussed extensively in other documents [4,9] and will therefore only be briefly reviewed in this section. More attention will be given to concepts specific to the SFEP and to definitions of terms.

SECURITY POLICY

The security kernel within a system is that combination of hardware and software that enforces the security policy. The definition of a security policy involves a description of the controls and rules that must be enforced when a subject accesses an object. The military security policy is defined with respect to a person's access to data, where each person (subject) is assigned a clearance and each data item (object) is assigned a classification. In addition, subjects and objects may be assigned to one or more categories or compartments that further define or limit the available access. The concept of clearance, classification and category has been generalized for use with the security kernel. The term "access level" is used as a replacement for clearance, classification, and other security attributes of people and objects. The military policy, called the "simple security condition", states that an individual may not see data of an access level greater than his own. Within the computer, the active agent for a person is a process, and the process acquires the security attributes of its user. The computer system must then make sure that this simple security condition is satisfied whenever a process accesses an object.

In the real world, people are "trusted" not to violate the policy with respect to data to which they have rightfully been given access. For example, a secret cleared individual is trusted not to disclose secret or confidential information to someone not properly cleared. He is not trusted, however, to refrain from looking at or disclosing top secret material.

Within the computer, however, the process that acts on behalf of the individual is not trusted to prevent a compromise of data, even if it has access to the data as indicated by its security level. This lack of trust in software has led to the requirement for a protection policy more restrictive than that enforced for people. The additional restriction placed on software is that a process cannot "write" into an object of an access level less than its own.[1] This restriction is commonly known as the *-property [8], and when combined with the simple security condition is the policy that the kernel must enforce.

SECURITY AND INTEGRITY

The "access level" attribute that is assigned to processes and data is more complex than that alluded to above. An access level is actually made up of two identically structured components: a security level and an integrity level. The security level is used to protect against the unauthorized disclosure of information, as required by the Department of Defense regulations. It directly embodies the familiar classifications and categories assigned to individuals and to information.

The integrity level is used to protect against unauthorized modification of information. The concept of integrity was introduced for use in computer systems [10] because, using only security levels, the problem of sabotage of classified information is not addressed. For example, security levels allow an unclassified program to be used by a top secret process. If this unclassified program is untrustworthy, there is nothing to stop it from maliciously destroying or modifying top secret information to which the process has access. By assigning an integrity level to a process and to programs and data, a process of high integrity will not be allowed to access data or use programs of a lower integrity, thereby preventing sabotage by programs of lower integrity.

Both the integrity level and security level are made up of two components: classification and category. The classification is a single value, and the category is a set of values. Classifications are linearly ordered and category sets are partially ordered. In comparing two access levels, A and B, one of several relationships can hold:

$A = B$
$A \neq B$ and A "dominates" B
$A \neq B$ and B "dominates" A
none of the above: A is isolated from B

[1]Strictly speaking, there are not always simple "less than" or "greater than" relationships between access levels. These terms are used, however, when the meaning is obvious. See the next subsection for a discussion of the structuring of an access level.

The rules for determining when the "dominates" condition is satisfied are presented in the specification of the Dominates function in Figure 13. A process of access level A can only read an object of access level B if A dominates B. A process can only write an object if B dominates A. If A dominates B and B dominates A, then A must be equal to B and thus the process may both read and write the object.

KERNEL CONCEPTS

The kernel, as discussed above, is a combination of hardware and software. The kernel must be as small as possible, isolated, and tamperproof. In a computer such as the SFEP or Multics, the kernel software is envisioned to run distributed in every user process. Isolation is provided by putting the kernel in the innermost ring or rings protected by hardware [11]. All kernel software and hardware must be completely and formally specified. Software in a ring outside the kernel invokes the kernel by calling specific entry points that in the specification are known as "primitives". From the point of view of uncertified software, these primitives are indivisible. The top level specification, presented in this report, is a specification of the functions provided by these primitives in terms of effects visible to non-kernel software.

The kernel itself may be internally layered in order to simplify verification. Layering of the kernel itself, and the functions of each of the layers, are topics to be considered in the future. It is sufficient to say at this point that the top level interfaces entirely specify the operation of the kernel from the point of view of non-kernel software.

Since the kernel only provides the most basic functions necessary to build a secure system, it is envisioned that a supervisor of some form will run in the rings immediately outside the kernel. This supervisor must be itself protected from user software that runs in the outermost rings, but such protection is only necessary to provide a functional utility -- not security. As far as the kernel is concerned, there is no difference between supervisor and user software.

It is likely that only the supervisor will directly call kernel primitives. By monitoring all kernel calls, the supervisor is able to perform its necessary functions and keep the system running. It is hoped that the kernel provides sufficient generality to allow the efficient implementation of a wide range of operating systems. However, discussion of possible supervisor or applications software design is outside the scope of this report.

SECURITY CONTROLS

The security policy enforced by the kernel consists of two basic types of controls -- discretionary and non-discretionary. These two types of controls are the only ones that the kernel provides. Other controls, such as those enforced by the ring mechanism, are not provided by the kernel.[2]

Non-discretionary Security

The simple security condition and the *-property comprise the non-discretionary security controls. These controls are termed non-discretionary because non-kernel software cannot manipulate the attributes of subjects or objects upon which these controls are based. For example, a secret process can only create secret segments. Secret segments, once created, must remain secret -- no process has the capability of changing the access level of a segment, either up or down.

Discretionary Security

Discretionary security is derived from the Department of Defense need-to-know policy. In the real world, an individual does not have the right to change the classification of a document, but he does have the right to provide access to the document to other properly cleared persons if they have a need-to-know. In the computer, need-to-know controls are provided by use of an access control list (ACL) for each object. The access control list specifies who can access the object, subject of course to the non-discretionary controls discussed above. A process that creates an object can set the ACL as it desires. The kernel places no restrictions on how the ACL can be set, but the ACL is saved within the kernel and is enforced by the kernel when another process attempts to obtain access to the object.

There has been a certain amount of controversy over the usefulness of discretionary controls implemented by the kernel because, by their very nature, they cannot be trusted. The fact that the controls are discretionary means that any software running in a user´s process

---

[2]The use of rings to protect the kernel from user software is not an issue at the top level interface -- the kernel simply does not provide any primitives that allow the user to set the ring numbers of segments to kernel ring numbers. There are functions provided by the kernel that set ring numbers in segments for users, but these functions appear only in the implementation and not in the specification because they have no "net effects". See the discussion of net effects beginning on page 17 and particularly the example of an unspecified effect near the middle of page 19.

can maliciously or incorrectly change ACLs of objects he owns without the user's knowledge. In order to "prove" that ACLs correctly reflect the user's wishes, it is necessary to verify all the code to which the user has access, in addition to the kernel code. Since verification of non-kernel code is infeasible, what reason is there for providing a part of the ACL controls in verified kernel software when non-verified code can easily subvert it?

The answer to this question lies in the definition of the security policy. The policy states that if need-to-know restrictions are properly specified, they are enforced. The proof of ACL controls will concentrate on correct maintenance of the ACL at the kernel interface (assuming that the arguments passed to the kernel are correct), proper enforcement of the ACL, and on correctness of the implementation as specified. The utility of the ACL controls is that, if it can be shown that the ACL setting primitives are correctly called, the ACL is properly enforced.

MINIMIZATION OF THE TOP LEVEL

The top level specification of the kernel is strictly defined to contain those interfaces that are visible to uncertified or non-kernel software. The goal of the top level specification is to include only those functions that are necessary to enforce the security policy, and to minimize the complexity and size of those functions. In the process of designing the top level interfaces, however, it often is necessary to include functions that may not be directly related to the enforcement of security but are required to provide a desired feature or avoid redundancy. Also, the ability of the supervisor to operate efficiently and reliably must often be considered.

As examples of functions not required for security, consider those functions that return the status of objects (such as Get_acl shown in Figure 14 and Get_attributes in Figure 17). These functions are not necessary for security because the supervisor, who creates objects via kernel calls, can keep its own records of the objects. However, a system crash or supervisor bug can destroy such records, and it should be possible for the supervisor to recover. (Of course, it is assumed that the kernel has no bugs and can itself recover from a hardware crash.) Also, since the kernel must save these attributes, it would be unnecessarily redundant for the supervisor to also keep such records. The status functions are provided to fulfill a utility requirement and to aid efficiency.

It is not always completely clear whether a given function should be provided by the kernel. If there is any doubt about inclusion of a function, the goal of minimizing the kernel dictates that that function be left out unless its impact on the specification is trivial. In the Multics design, for example, the Multics quota mechanism is enforced by the kernel because the mechanism is part of Multics that must remain compatible and because there is no way for the supervisor to provide that same quota mechanism without help from the kernel. In the SFEP design, compatibility is not an issue and, since the supervisor could implement some form of quota mechanism if necessary, no quota mechanism is provided by the SFEP kernel.

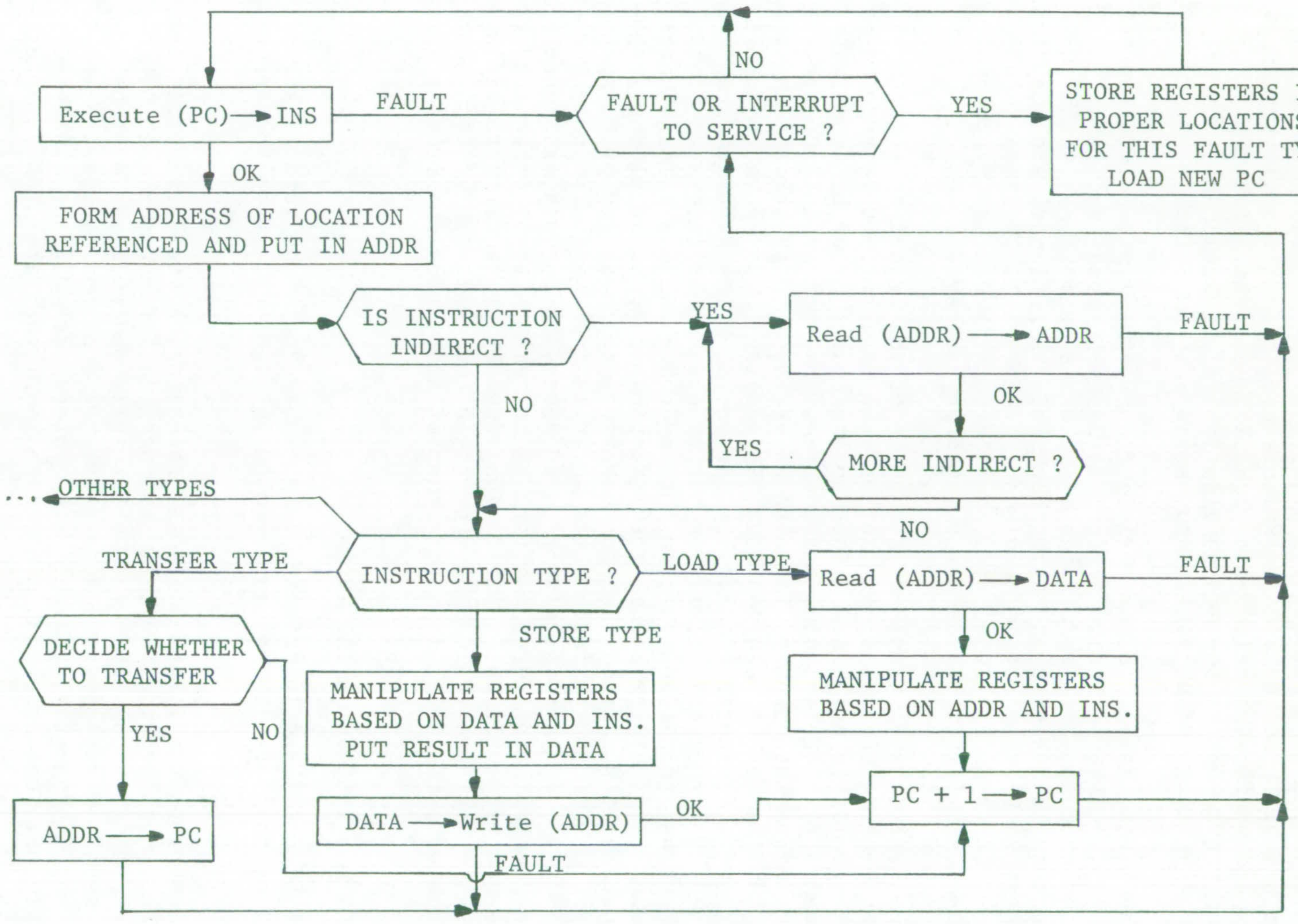
THE INTERPRETER CONCEPT

The "interpreter concept" is employed as a means by which one can visualize the operation of a kernel based machine. The top level kernel specification completely defines an abstract machine and all the available operations. The interpreter is an abstract program, outside the kernel, that runs on this abstract machine. The interpreter itself can provide a series of interfaces that define a further outer layer of abstract machine, upon which another abstract program can be constructed, and this layering can be continued indefinitely outward in a manner similar to the layering of abstract machines within the kernel.

The Hardware Interpreter

In order to support an operating system or general user programming outside the kernel, some layer of abstract machine outside the kernel must present all the interfaces normally available to programs, i.e., the complete set of hardware instructions and facilities must be provided, in addition to the kernel´s software implemented functions. Because this abstract machine is not in the kernel, its exact makeup is not important. Thus it can, for simplicity, be assumed that the interpreter running immediately outside the kernel is the abstract program that, using only kernel functions, presents a complete hardware and software interface at the next layer of abstract machine out from the kernel.

This interpreter can be likened to a microprogrammed emulator that provides for the execution of complex machine instructions using a basic set of primitive mechanisms. A rough idea of what such an emulator might look like is shown in the flowchart in Figure 1. The figure is merely illustrative and very general in nature -- it is not meant to depict the detailed instruction cycle of the Level 6 minicomputer. The interpreter has an internal data base that is used to hold various flags and registers needed to emulate the instruction set. This data base includes such things as the program counter (PC in the flowchart), instruction word (INS), address register (ADDR), temporary



Execute (PC) ⟶ INS
FAULT
FAULT OR INTERRUPT TO SERVICE ?
NO
YES
STORE REGISTERS IN PROPER LOCATIONS FOR THIS FAULT TYPE LOAD NEW PC
OK
FORM ADDRESS OF LOCATION REFERENCED AND PUT IN ADDR
IS INSTRUCTION INDIRECT ?
YES
Read (ADDR) ⟶ ADDR
FAULT
OK
NO
YES
MORE INDIRECT ?
NO
... OTHER TYPES
TRANSFER TYPE
INSTRUCTION TYPE ?
LOAD TYPE
Read (ADDR) ⟶ DATA
FAULT
OK
STORE TYPE
DECIDE WHETHER TO TRANSFER
MANIPULATE REGISTERS BASED ON ADDR AND INS.
MANIPULATE REGISTERS BASED ON DATA AND INS. PUT RESULT IN DATA
YES
NO
PC + 1 ⟶ PC
DATA ⟶ Write (ADDR)
OK
ADDR ⟶ PC
FAULT

Figure 1. Interpreter Flowchart

data word (DATA), and some state flags. Machine instructions are fetched from memory one at a time using the kernel's Execute primitive, decoded, and executed. The execution of a machine instruction may include calls to the Read or Write kernel primitives, setting registers, and performing arithmetic operations on data. In addition, the interpreter checks for and handles faults and interrupts at the appropriate time, and properly manages the program counter.

The kernel itself is responsible for providing all primitives required by the interpreter to perform security-related functions. Thus, for example, the Read, Execute and Write primitives, which involve access to segments and thus checks of discretionary and non-discretionary security, must be provided by the kernel. Arithmetic operations, transfer operations, etc., can be totally handled by the interpreter using its local data for temporary storage.[3]

The syntax, semantics, and structure of the actual code in the abstract program comprising the interpreter is totally unrestricted as far as security is concerned. Thus no one need ever be burdened with the massive task of designing and implementing such an interpreter. However, since the hardware implemented by the interpreter and the kernel is fixed and already defined, it remains to be proved that all security-related functions actually performed by the hardware are specified as kernel primitives, or can be decomposed into a combination of kernel primitives that an interpreter could use. This "proof of completeness" of the kernel specification could be obtained by providing an existence proof that the hardware, as specified by the manufacturer, can indeed be implemented in an interpreter using only kernel primitives and non-security related operations. This existence proof may entail considerable effort -- it remains to be determined to what degree of detail the verification methodology requires specification of the interpreter in order to obtain the existence proof. Hopefully the effort will be less than that of specifying and proving a kernel that includes all of hardware.

Interpreter Extension to Software

In Figure 1 the interpreter is shown to "call" only those kernel primitives that are in fact implemented in hardware. Upon further ex-

---

[3]Note, however, that, since all data repositories are state variables, the kernel must provide for their storage via V-functions. Thus, interpreter data, though it may be arbitrarily manipulated, must be included as a V-function (with appropriate O-functions to set its value). Read_interpreter_data and Write_interpreter_data are the abstract kernel functions that manipulate the interpreter data (See Figure 19).

amination it becomes evident that this abstract interpreter must also call functions that will be implemented in kernel software. Consider a software call to a kernel function. This call will appear, at the machine instruction level, like a call instruction with an appropriate argument list.[4] Since the interpreter provides the "complete" machine interface visible to non-kernel software, it is the interpreter that must decode the call instructions and argument lists for all calls that software makes to the kernel. Presumably at some point in the decoding of the call instruction the interpreter must exit to the proper kernel procedure, at which point it relinquishes control of the machine to the kernel.

Note that, since the abstract interpreter is outside the kernel, any argument validation that takes place as part of the decoding of the call instruction must be considered to be done by the kernel.

Obviously, as far as the hardware is actually concerned, the only difference between non-kernel software and kernel software is the fact that some protection device, such as a ring number in a hardware register, indicates a certain state. It is in reality the same hardware that executes kernel code. In the abstract sense, however, we have no choice but to assume that the kernel software somehow executes correctly, even though we don't care about the details of the hardware interpreter that "runs" non-kernel software.

NET EFFECTS

Because there is no well defined isolation mechanism that separates the non-kernel portions of the hardware from the kernel portions, we are at freedom to define the simplest set of primitives that implement the security-related hardware operations, and to consider all the rest of hardware to be in the non-kernel interpreter as discussed above.

Kernel software, on the other hand, will be isolated from non-kernel software by hardware-implemented rings (where the innermost ring or rings are occupied by the kernel). There is thus a well defined boundary between kernel and non-kernel software, and it becomes a simple matter to determine, after it is implemented, whether a given piece of code is inside or outside the kernel. It would appear, then, that if the specified top level kernel primitives were designed to be

---

[4]Call and return instructions are special hardware instructions that provide for ring crossings within a process. The Level 6 SPM supports a simplified version of the call and return mechanism as implemented in Multics [11].

minimal, there would be no problem constructing minimal software that directly implements these primitives in the kernel rings.

Unfortunately, the implementation is never as "clean" as the specification, and there is a general problem a verifier faces when it becomes necessary to prove the correspondence of the implementation to the specification. Although the specification itself may be complete, the goal of minimizing the top level results in the omission of certain important effects from the specification that are required only as implementation details. Such effects may even be visible at the top level. For example, it is quite possible that for each function, the kernel will always clear out one or more of the machine registers (which are part of the interpreter data) upon returning to the caller. An implementation detail such as this is left out because it would only clutter up the specification. The verifier, however, must deal with this and similar effects in some manner.

It may be a simple matter to prove that the zeroing of registers, providing it is always done, is secure, but more complicated effects may be more difficult to eliminate. The verifier is thus faced with a kernel implementation that appears to do much more than it is supposed to, according to the specification. In order to verify that the enlarged implementation of the kernel still corresponds to the specification, it is necessary to examine the implementation to show that any "extra" things done by the kernel have no net effect not specified at the top level. A net effect is an effect visible to the user that cannot be duplicated outside the kernel using existing kernel primitives.

The simple example of register zeroing can be duplicated outside the kernel by appending to all kernel function calls a call to the V-function that writes the interpreter data. In the search for net effects, one does not proceed on a function by function basis, since it is possible for the implementation of a given function, when examined separately, to have an unspecified effect. Rather, it is necessary to consider the entire specification and its implementation as a whole in order to determine whether there are any net effects.

The manner in which unspecified effects are checked during the validation procedure is straightforward. For any unspecified effect performed by kernel software, an example must be created that exactly duplicates the effect in all cases, in terms of variables and data to which the user has access, using only specified kernel functions. If such an example can be found, the effect is not a net effect and can be ignored. This existence proof is usually very simple, since the vast majority of unspecified effects are entirely process-local involving data of the same access level as the process. Only one "example" of the writing and reading of such data needs to be created to suffice for the existence proof of most instances of unspecified effects.

The purpose of this report is not to discuss validation techniques. It may very well be that net effects, as discussed here, will be treated in some entirely different manner. The purpose of discussing net effects at this point is to provide guidelines and a justification for not including in the specification certain features that, at first glance, appear to be required kernel functions because they will be performed by software running in the kernel ring.

Occasionally the specification contains what may seem to be redundant effects in the sense of net effects discussed above. Such effects are specified because the function provided is too deeply tied in with the operation of the kernel to be easily removable. The removal of such a function would leave a specification that is so abstract that proof of correspondence of the implementation might be extremely difficult. It is also possible that the inclusion of a "redundant" function actually simplifies the specification by localizing a series of exceptions or effects that might otherwise be dispersed throughout many functions. The Initiate primitive, and the entire concept of segment initiation, is an example of a redundant function whose removal might seriously confuse the specification and hamper the correspondence proof.

Example of an Unspecified Effect

An example of a facility provided by the kernel, but not specified, is the facility of storing ring numbers in segment descriptors at the time a segment is initiated. Rings, though they provide protection for the kernel itself, need not be supported to satisfy any security policy with respect to non-kernel data. There _is_ a functional requirement that the supervisor be able to protect itself from user code, and rings are used to provide this protection, but the requirement is imposed on the implementation and not on the specification. There is no inherent reason that the kernel itself provide support for rings at the top level.[5]

Unfortunately, due to the hardware architecture, the only place ring brackets can be stored is in segment descriptors, and segment descriptors must be physically within the kernel domain because they are the main protection element for data. Therefore, the kernel implementation must provide a function that inserts these ring brackets, at

---

[5]The fact that the same mechanism used to isolate the kernel is also used to isolate the supervisor is irrelevant in the abstract sense. Verification of the implementation guarantees that only kernel segments have kernel ring numbers.

the discretion of the supervisor (or interpreter), into the segment descriptor. Such a function is said to have no net effect, however, and therefore need not be specified at the top level, because, from the point of view of the abstract machine provided by the kernel, this function does nothing that cannot be duplicated by the interpreter or supervisor using existing kernel primitives. (The only effect of setting a ring number in a segment descriptor is the possibility of subsequently generating a fault when that segment is referenced in the same process. This is tantamount to setting a flag for oneself and later testing it -- an operation easily performed using process-local interpreter data.) Specification of such a function would be entirely redundant, contingent, of course, upon the ability to prove that ring numbers in descriptors have no other security-related effects.

For the most part, implementation details that can be said to have no net effects are those that only involve manipulation of process-local data. It is possible, however, that a function can perform an operation that is visible to other processes of the same security level only and may therefore still have no net effect.

TRUSTED FUNCTIONS

Since the policy enforced by the kernel is only specified in terms of access to data by untrusted users and software, the kernel is not responsible for maintaining any controlled interface to trusted users or trusted software. The top level specification of the kernel only refers to the interfaces to untrusted software and untrusted users.

No system can properly perform, however, without a certain amount of software that must be trusted to work correctly, even though that software is not responsible for enforcing any security policy. For example, a "downgrade" function may be provided that allows certain privileged individuals to downgrade classified information. The primitive functions that provide such interfaces to trusted individuals are termed trusted functions.

Trusted functions are not strictly part of the top level kernel interface because the top level interface as a whole enforces the policy regardless of how it is invoked. The verification of the security kernel involves proving that the specification enforces the policy, and that the implementation of the specification is correct. Trusted function interfaces, on the other hand, will violate the policy if they are invoked incorrectly.[6] Under certain conditions it may be

[6]Note that if the security policy is changed to include the operation

possible for a trusted function to make certain security or consistency checks on the arguments passed to it, thereby providing some assertions upon which to base a proof of the specification, but in general the only thing that can be proved about a trusted function interface is that the implementation agrees with the specification.

Though they are not part of the top level interface to the kernel, the trusted function interfaces must be specified because they are interfaces to the outside world. In this report trusted functions are discussed in Section VII, and the specification of the interfaces is presented. It must be understood, however, that they must be invoked by trusted users (<u>not</u> by untrusted software).

SFEP KERNEL DESIGN OVERVIEW

The SFEP kernel is designed to be general purpose in that a wide variety of operating systems can be supported. Conceivably the kernel could be somewhat simpler if only the specific Multics SFEP application were considered, but it is difficult to determine in advance exactly which features ought or ought not be provided by the kernel for a specific application. The task of validation of the kernel is great enough so as to warrant inclusion of features of a general nature, rather than specialized features that may later have to be modified and recertified. In addition, a generalized kernel does not appear to be that much more complex than any envisioned specialized kernel such that the validation effort would be significantly affected.

The overall design of the SEEP kernel is very similar to that of the Multics kernel. At the top level, the kernel supports a file system structured as a hierarchy with a full set of primitives for manipulating objects in the file system. The process structure is similar to that of Multics, with block and wakeup primitives for interprocess communication. The kernel also supports user I/O, a capability made possible by Level 6 hardware-implemented access control features for I/O devices.

The top level specification is presented in three parts -- storage control, process control, and input/output -- in the following three sections. Section VII discusses the trusted functions. Appendix II contains an index of individual function names and the figures in which they can be found.

of these trusted functions, the trusted functions then become part of the top level interface to be verified, and need no longer be considered trusted functions.

SECTION III

SPECIFICATION LANGUAGE

The top level specification is a formal description of a finite state automaton that implements the kernel interface of the SFEP. This interface enforces the access control axioms developed in the mathematical model.

BACKGROUND

The requirements of verification imply a need for a rigorous and unambiguous specification. The specification provides the desirable explicitness by adhering to formal specification standards and techniques described below.

The form of the specification is derived from a technique developed by Parnas [12] and extended by Price [13]. Additional extensions and modifications have been incorporated specifically for application to the Multics and SFEP kernels.

SPECIFICATION FUNCTIONS

A "Parnas specification" provides a method of describing a finite state automaton. The machine state is embodied in a set of "value" functions (V-functions) that return a value when invoked. All possible changes to the machine state are described by a set of "operate" functions (O-functions) that perform operations on V-functions based on the values of supplied parameters. A third set of functions, OV-functions, effect an operation and return a value. OV-functions are used when it is necessary to make an operation indivisible from the return of a value.

There are two classes of V-functions, derived and primitive. Primitive V-functions are the state variables that can be assigned values and can be invoked to obtain values. Derived V-functions can be invoked to obtain values that are totally derived from the values of other V-functions. As such, derived V-functions cannot be "set" to a value but can only yield values. Derived V-functions are not required to describe the machine state, since they do not represent the values of independent state variables, but they are used to return information derived from state variables that would not otherwise be visible outside the kernel.

The kernel, as a finite state automaton, is specified by a set of O, V, and OV-functions that are visible to and may be arbitrarily invoked by non-kernel software, hardware, or users. These functions define the kernel interface and make up the top level specification that will ultimately be validated with respect to the mathematical model.

In order to allow for the specification of machine state variables that must not be visible outside the kernel, or must not be arbitrarily set outside the kernel, but which are necessary to fully represent the kernel´s internal state, hidden V-functions are provided. Hidden V-functions are similar to non-hidden V-functions but can only be invoked from within the kernel (by other kernel functions). In practice, because most state variables may not be arbitrarily set outside the kernel, primitive V-functions are almost always hidden. Even if there is a simple one-to-one mapping between the values of a primitive V-function and a corresponding derived V-function, it is usually necessary to make the primitive V-function hidden and to define a non-hidden O-function that sets its value in order to make appropriate checks as to when the value may be set.

There are two types of "functions" that are really macros used to simplify the specification. They are O-function macros and V-function macros, syntactically similar to O-functions and derived V-functions, respectively. These macros are understood to be invokable only by other kernel functions, and can thus be considered hidden. O-function and V-function macros generally specify operations or values that may be common to several O-functions.

EXCEPTIONS

In the specification of most non-hidden functions there are a series of exceptions that are checked, in a specified order, before the operation is invoked or the value is returned. These exceptions are as important to the specification as the effects or values returned, because they protect against unauthorized modification of data or inconsistent machine states. In addition, because the caller is informed of which exception occurred, exceptions are used to communicate information back to the caller for which, if there were no exceptions, additional V-functions might have to be provided to supply this information.

Since exceptions return information, the top level proof of the specification must consider whether improper information is returned by the exception. In many cases, the order in which the exceptions are checked and signalled is critical as to whether illegal information is transmitted to the caller. Also, in some cases the ability to distinguish between one of two logically different exceptions may provide a basis for obtaining illegal information. In such cases it is necessary to combine the two exceptions into one such that there is no way to tell which one occurred.

Exceptions may only appear in non-hidden O-functions and V-functions. Hidden functions cannot have exceptions, since they must be invoked "correctly" by the kernel. Therefore, hidden V-functions must be defined for all possible values of the input parameters that the kernel might pass as arguments. Of course, function macros cannot have exceptions either, since they are merely used as a form of shorthand notation.

SPECIFICATION SYNTAX

Appendix I describes the complete syntax of the specification. Within this document, the specification is presented, not as a whole, but broken up into pieces and placed into figures for readability. Since this breaking up results in partial or incomplete components of the specification appearing in various places, the syntax rules may not appear to be precisely followed in the figures. The syntax described in the Appendix is, however, a syntax that could be used for the validation procedure when all levels of the specification are finalized.

As a supplement to the specification syntax, some highlights of the specification language and rules are discussed below, along with some discussion of the semantics of the specification.

The specification is composed of a set of modules, each of which contains the five sections: module name, type, define, parameter, and constant, followed by individual function specifications. The entire SFEP specification is considered to be one module.

Module Sections

The five sections of the module contain information that is global in nature and applies to all function specifications. A description of each of the five sections follows:

name:	This section simply contains the name of the module.
type:	The type section supplies global type definitions that declare the types of all parameters and V-function values. Strong typing is provided, thereby simplifying the specification by eliminating the need for explicit bounds checking in the functions. The syntax of the type specifications is taken from Pascal [14]. There

are three primitive types, not explicitly defined in the specification, as follows:

boolean: scalar ("true", "false");
character: scalar ("A", "a", "B", "b", ..., "Z", "z");
integer: scalar (-2^{35}, ..., -1, 0, 1, ..., $2^{35}-1$);

More complex types are built up from these primitive types and other types using the following type constructors:

scalar: one of an ordered set of values.

vector: a fixed length list of components of a specific type. A component of a vector is obtained via a selector having a specified range or type. A reference to an element of a vector is written as the vector name followed by the selector value in square brackets, e.g., "level[i]".

structure: a fixed length set of components, each having a name and type.

set: a variable length set of components all of the same type.

define: The define section provides global definitions for certain identifiers. These definitions are simple direct substitution macros that allow for shorthand notation for a specific string of text. Defines can be removed from the specification with no effect by substituting each occurrence of the identifier in the specification with the appropriate text.

parameter: All formal parameters and quantified variables used in functions are declared in the parameter section. Each parameter is given a type definition, thus providing the strong typing of function parameters. Quantified variables, usually used as indices in local "lets" (see below), have a range as indicated by their type definition.

constant: The constant section defines named constants and their types whose specific values are of no concern in the specification. Constants have a value that must be invariant in the life of the system.

Function Sections

Each function specification is broken up into several sections, most of which are optional. These sections are defined as follows:

name: The function name appears on the first line of the function specification, followed by a parenthesized list of parameters, and the type of its value if it is a V-function or OV-function.

let: The let section contains local macro definitions that apply only to the function. The syntax and semantics are identical to that of the define section of the module described above. When substituting macro strings, local "lets" are to be expanded first, before scanning for any "defines".

exception: The exception section contains a list of boolean expressions that are evaluated in order and cause a return to the caller if any evaluation yields the value "true". An indication of the first expression yielding "true" is also returned. Exceptions may only appear in non-hidden functions.

effect: This section appears in O and OV-functions. It contains a list of boolean expressions describing the state transitions that take place when the function is called. The order of evaluation of the expressions is not important -- the final state must be the same regardless of the order of evaluation of the expressions. Any V-function referenced in an effect represents the value associated with that V-function after completion of all the effects. If a V-function is referenced as a value in an effect, and is also assigned a new value by the same function, all references to the previous value of the V-function are preceeded by a single quote (´). Because effect expressions are simply assertions, the two expressions:

```
                 3 = x;
and              x = 3;
```

are equivalent, i.e., the statement that "x is equal to 3" is "true" at completion of the function. In this example, the only way to make the statement true is to assign the value 3 to the variable (or V-function) x. For more complex expressions, it may not always be obvious which V-functions are "set" in order to make the statement "true". By convention, in order to avoid ambiguity, expressions in effects are, if possible, written as:

```
<identifier> = <expression>
```

where it is assumed that the <identifier> is assigned the value of the <expression>. In some cases, however, as in the specification of Block in Figure 21, the expression cannot be written as an assignment -- its meaning, in such a case, is obvious.

derivation: The derivation, which appears only in derived V-functions, V-function macros and OV-functions, contains one or more expressions of the form

```
<identifier> = <expression>;
```

where <identifier> is either the name of the V-function or, if the V-function value is a structure, the structure component name.

"Undefined" Values

Normally, a primitive V-function has a value of a specific type, as last set by some O-function. In addition, any V-function, or component of a structure in a V-function, may have the value "undefined", regardless of its type. The "undefined" value is implicitly assigned to any V-function whose value has not been set, and may also be explicitly assigned and tested for. If a structure is set to "undefined", each of the components is also set to "undefined". If a component of such a structure is subsequently set to some value, the remaining components of the structure remain "undefined", but the structure itself will no longer be equal to "undefined". By convention, it is not permissible to reference an "undefined" value, except as a test for equal or not equal to "undefined". Note, however, that in an expression such as:

```
a = 5 ¦ b = 6 ¦ c = 7
```

it is permissible for both a and b to be "undefined", as long as c = 7. That is, if the value of the expression can be determined without evaluating any of the undefined quantities, the expression is permissible. In the above example, if $c \neq 7$, the value of the expression is not deterministic and expression is semantically in error.

Expressions

The syntax of expressions is not precisely defined so that their expressive power is not limited. The assumption is that well known arithmetic and mathematical symbols will be used so that the meaning of an expression is apparent. If any "strange" symbol is used, a comment should appear in the specification stating its meaning. For the SFEP, a certain limited set of operators and constructs has been chosen. The description of each operator below includes the function of the operator, acceptable types of the operands, and type of the result of the operation. The notation uses a meta-language described in the introduction to Appendix I.

Prefix operators:

-	arithmetic negation -- integer operand and result
^	logical complement -- boolean operand and result

Infix operators:

+ - / *	arithmetic plus, minus, divide, multiply -- integer operands and results
& |	logical AND, OR -- boolean operands and results
> ≥ < ≤	comparison operators -- scalar operands; boolean result
= ≠	equal; not equal -- operands may be of any like types; boolean results
∈	element of set -- operator on left is a scalar, right operand is a set of same type; boolean result
∋	such that -- see complex constructs below for description.
||	concatenation -- boolean vector operands; boolean vector result.

Complex constructs:

```
if <boolean expression> then <expression>+
                  [else <expression>+] end
```

This is a conditional evaluation of the appropriate <expression>s depending on the value of the <boolean expression>. If used to return a value, the value of the appropriate expression is returned.

(∃<parameter>+)(<boolean expression>)

This construct yields the boolean value "true" or "false", depending on whether there exist values of the <parameter>s such that the <boolean expression> is "true". The range of values of the <parameter>s is implicit in the type of the <parameter>, except that those values of the <parameter> that would make the <boolean expression> non-deterministic (due to improper referencing of "undefined" values as discussed on page 27) are excluded. For example,

```
    (∃iuid)(Process(iuid).uid = uid1)
```

is evaluated by examining the value of Process(iuid).uid for all possible values of the parameter iuid. In doing so, it is likely that a reference will be made to "undefined" values, which is illegal. The expression could be rewritten as

```
    (∃iuid)(Process(iuid) ≠ "undefined" &
         Process(uiid).uid = uid1)
```

which is more precise. However, in order to simplify the specification, the semantics of this construct have been defined so that the explicit check of "undefined" is unnecessary.

(∀<parameter>+)(<expression>+)

This construct yields the boolean value "true" if the <expression>s are "true" for all values of the <parameter>s. If used in an effect, this construct results in evaluation of the <expression>s for all values of the <parameter>s.

{<parameter>|<boolean expression>}

This construct yields a set consisting of all values of the <parameter> such that the <boolean expression> is "true".

<parameter> ⋟ <boolean expression> or
(<parameter>+) ⋟ <boolean expression>

This construct yields any single value of each of the <parameter>s such that the <boolean expression> is true. If more than one <parameter> is specified, this construct must be used in an assignment as follows:

(<parameter>+) = (<parameter>+) ⋟ <boolean expression>

which causes assignment of each <parameter> on the left side to the corresponding <parameter> on the right side, respectively. In the example

uid = iuid $\ni$ Process(iuid) $\neq$ "undefined"

the parameter uid is set to any value of iuid such that Process(iuid) is not "undefined". In the construct

(proc,device) = (iuid,idevice) $\ni$ <expression>

the parameters proc and device are set to a value of iuid and idevice respectively, such that the <expression> is true.

Miscellaneous operators:

min {<set>} or min (<expression>+)

This yields the minimum of a set of scalar values.

max {<set>} or max (<expression>+)

This yields the maximum of a set of scalar values.

bit (<expression>)

This yields a vector of boolean types (string of bits) whose length and value uniquely identifies the value of the <expression>, which may be of any type. The actual mapping between values of various types and bit strings is of no concern to the specification.

SCOPE OF THE SPECIFICATION

The top level specification presents the kernel interface as it appears to a given process. This interface includes all process-local state information, the file system, and certain information about the existence of other processes. Since a process operates at a given access level, the process-local kernel interface can be validated with respect to the model in the sense that no information of a higher access level is returned to the process. In reality, though, the kernel is distributed among all processes, with each process executing as if it was the only one on the processor. In order for the system to be secure it is necessary to hide from a process the scheduling of other processes on a processor. The fact that there may be more than one processor in a system is also not visible at the top level. Thus, in order to provide such an appearance at the kernel interface, it is necessary to eliminate the notion of process switching at the top level. There is a hidden V-function in the specification called Cur_proc that provides the identifier of the process "currently" in execution, and its value never explicitly changes in the specification. It must ultimately be proved that the manner in which Cur_proc changes via some scheduling algorithm (either inside or outside the kernel) cannot be used to transmit information.

In one place in the specification, the Block primitive, a process explicitly gives itself up to the processor. However, Block is specified to simply "wait" for a specific event and return when the event occurs. As far as the process is concerned, no virtual time (i.e., processor time spent in the user´s process) has elapsed during the wait.

Because the kernel specification presents a per-process virtual environment, it is necessary to provide a mechanism for specifying operations that take place asynchronously. This is not a problem with Multics because all V-functions visible to the current process are changed by O-functions executed synchronously in some process. In the SFEP, there are asynchronous direct memory access (DMA) I/O devices that are outside the kernel and effect operations on V-functions visible to some process. These asynchronous devices may operate on behalf of a process other than the one currently on the processor. Thus, it is necessary to specify their operation using "asynchronous O-functions", which are understood to be invokable at any time by any device on behalf of any process. The kernel must determine which process is using the device in order to access the correct data bases, but Cur_proc must not be changed. In order to include asynchronous O-functions in the proof of the specification it will be necessary to make additional assertions about the manner in which they are called (e.g., that the device does not forge its physical identification).

SECTION IV

STORAGE CONTROL

Storage control in the SFEP is very much like that of Multics. The file system is a hierarchy as pictured in Figure 2. Each object or entry in the file system is either a directory, segment or device. Directories, indicated by rectangles, are directly accessible only to the kernel and are used to store identities and attributes of entries beneath them. Note that the attributes of a directory are contained in the parent directory. Segments are unstructured files directly accessible to the user, and devices are I/O devices, also directly accessible to the user. Though devices are objects in the file system, the Input/Output subsystem will be discussed separately in Section VI.

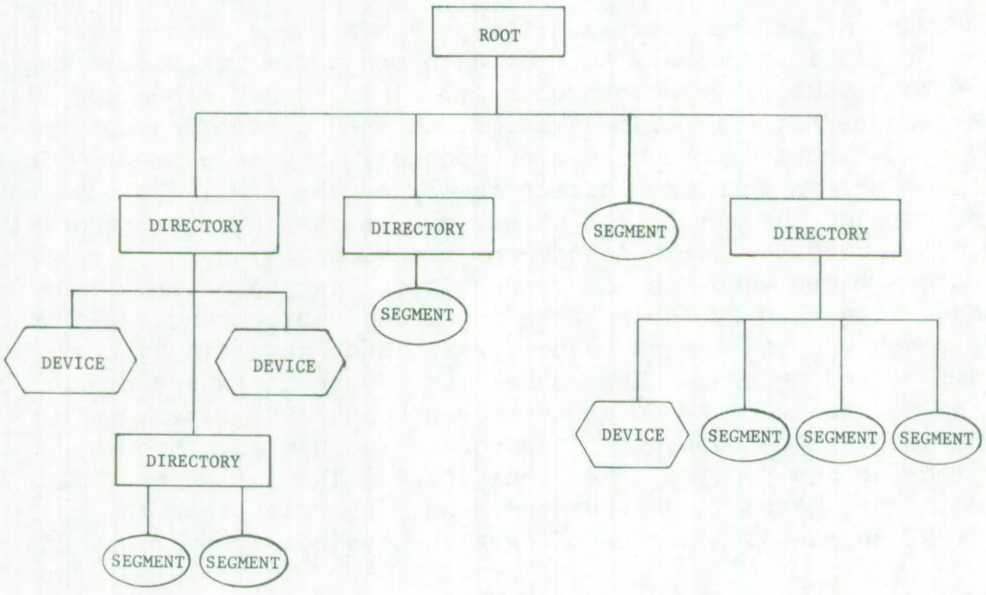


Figure 2. SFEP File Hierarchy

Every object (segment, directory or device) in the hierarchy has an access level that is equal to or greater than that of its parent directory. An object whose access level is greater than that of its parent is called an "upgraded" object. The access level of the root directory must always be the lowest possible (referred to as system_low).

As stated above, the attributes of an object are stored in the parent directory, and are thus classified at the access level of the parent. For upgraded objects, this means that the attributes are at a lower access level than that of the object and thus, because of the *-property, a process that can modify the object cannot modify the attributes. Similarly, a process at the access level of the attributes can modify the attributes but cannot examine the object itself. Upgraded objects can only be deleted by a process whose access level is equal to that of the parent.

VIRTUAL MEMORY LAYOUT

The structure of the virtual memory supported by the SFEP hardware is not actually relevant to the top level specification of the kernel. However, it is presented here because it is undocumented elsewhere and because the hardware structure does guide the kernel design -- even at the top level.

Virtual memory in the SFEP supports both memory and I/O devices as objects. The SFEP hardware performs the address translation and access checks necessary to access an object directly. Directories are software supported objects that cannot be accessed directly by non-kernel software, but they are nonetheless part of the virtual memory. Direct access to directories is allowed only in the kernel ring, and all requests for modifying or reading directories must be made via calls to the kernel.

Figure 3 shows how a virtual memory address, presented to the processor by a machine instruction, is translated into the physical address of the word to be referenced. The address translation mechanism is almost identical to that of Multics. For a complete description of the Multics mechanism, see [15]. A brief summary of memory address translation will be given here. Refer to Section VI for a discussion of device address translation.

At the top left of Figure 3, the descriptor base register (DBR), loaded by the kernel, is shown to point to the base of a descriptor tree for the current process. There are up to three levels of descriptors, corresponding to descriptor segment page descriptors, segment descriptors, page descriptors. The segment page descriptor points to the physical page in memory. At each level in the translation cycle, the previous descriptor is used to address the next descriptor table, and the appropriate few bits of the virtual address are used to index into this table to find the next level descriptor. The final few bits of the virtual address are an index into the page. The exact number of bits in each of the four fields of the virtual address is unimportant for the purposes of this discussion.

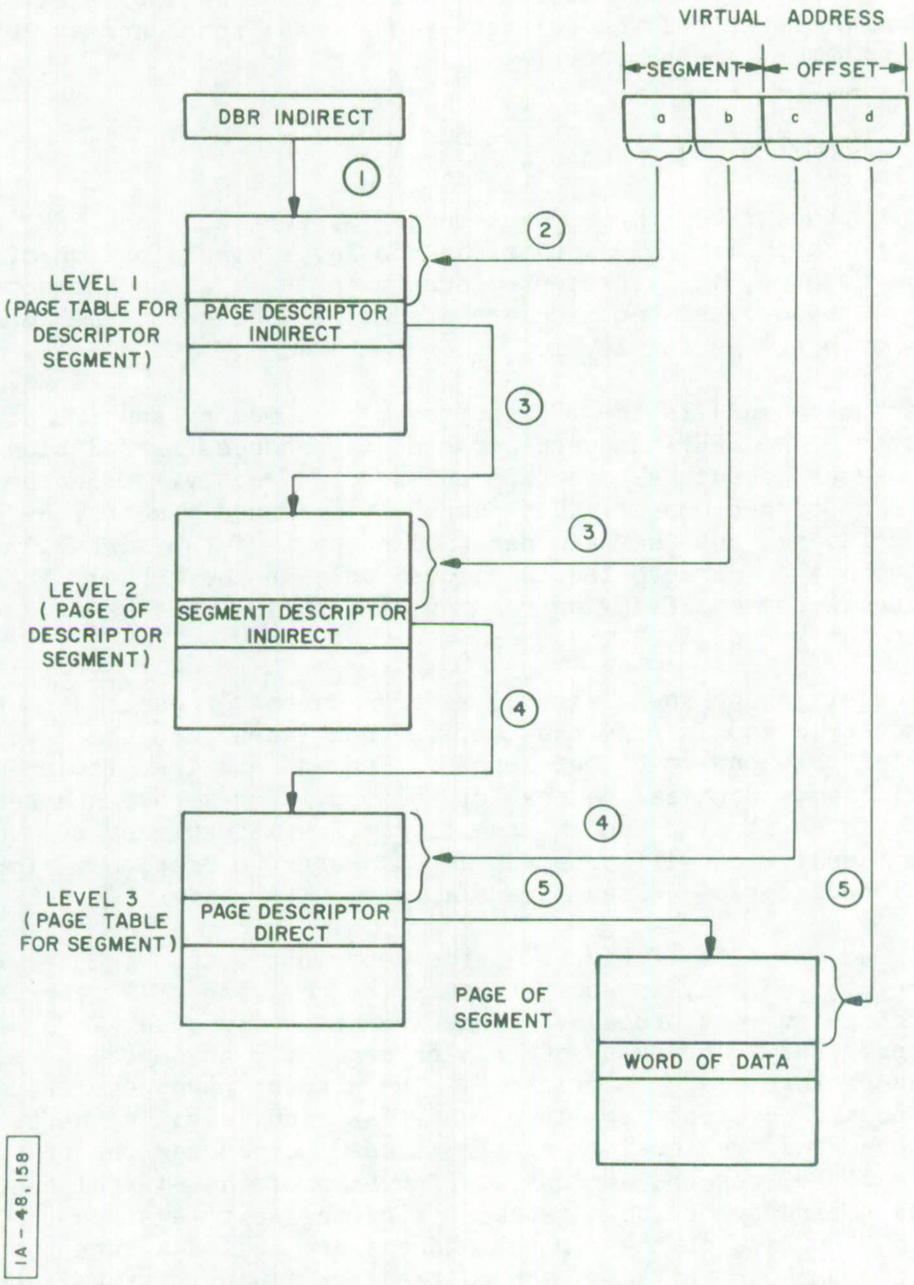

VIRTUAL ADDRESS
SEGMENT
OFFSET
a
b
c
d
DBR INDIRECT
1
2
LEVEL 1 (PAGE TABLE FOR DESCRIPTOR SEGMENT)
PAGE DESCRIPTOR INDIRECT
3
3
LEVEL 2 (PAGE OF DESCRIPTOR SEGMENT)
SEGMENT DESCRIPTOR INDIRECT
4
4
LEVEL 3 (PAGE TABLE FOR SEGMENT)
PAGE DESCRIPTOR DIRECT
5
5
PAGE OF SEGMENT
WORD OF DATA
IA - 48,158

Figure 3. MEMORY ADDRESS TRANSLATION

Access control information for each segment is stored in the segment descriptor, and applies to all pages in that segment. The access control information consists of the following:

rings:	Three ring numbers specify the domains in which access to the object (segment or device) are allowed. The usual meanings of the three ring numbers for access to segments are:

write bracket:	0 to r1
read bracket:	0 to r2
execute bracket:	r1 to r2
call bracket:	r1 to r3

For devices, the rings have the following meaning:

write bracket:	0 to r1
read bracket:	0 to r2
control bracket:	0 to r3

The read and write brackets refer to the reading and writing of data on the device, as opposed to transmission of control information (e.g., status or positioning requests). The "read only" mode of access (from rings r1+1 to r2) is a feature useful to software for write-protecting of the data medium. However, because control operations must always be "written" to a device, even if there is no accompanying writing of the data, all devices must be, for security purposes, considered to be read-write objects.

modes:	There are three mode bits associated with segments: r, e, and w. The modes are read, execute and write, respectively, and specify the type of access allowed provided the ring bracket restrictions are satisfied.

In a kernel based system, the above hardware controls are used to protect the kernel from the supervisor, and also are available for the supervisor and the user for further layers of protection. The kernel, most likely running in ring zero, will allow the supervisor no access to any of its segments (i.e., the kernel´s segments will have ring brackets of 0,0,0), except that entries into the kernel to invoke kernel primitives will be in gate segments whose ring brackets are 0,0,1. Thus the supervisor, running in ring 1, would be able to invoke the kernel, but the user, running in a higher ring, would not. Of course, the kernel and/or supervisor may occupy more than one ring, depending on the degree of layering desired. Also, there is no security-related reason for restricting kernel access to the supervisor. It is, however, unlikely that the supervisor could keep the system from crashing if it allowed unrestricted access to kernel calls. The SFEP hardware is specified to support a minimum of three rings -- one each for the kernel, supervisor, and user or applications programs. Additional rings may be provided, however, for either kernel or non-kernel use. At the time of this writing it appears that there will be four rings.

In an actual implementation the third level (page) descriptors will probably be shared by all processes using the page. In this way a page fault (page not in memory or page doesn't exist) can be signalled for all processes simply by marking the one page descriptor. The first two level descriptors, however, cannot be shared because they contain access control information that applies only to one process. Multics currently employs such a descriptor structure [16].

The descriptor tables themselves cannot be accessed by non-kernel software and their existence is in fact invisible at the top level. For this reason, the top level specification only deals with the file system and the visible kernel interfaces that support the virtual memory.

COMMON DEFINITIONS AND FUNCTIONS

In support of storage control and process control there are several data bases maintained by the kernel as hidden V-functions. Before discussing these data bases, it is appropriate at this point to define some of the data types and parameters used in the storage control specification. Following the type and parameter specifications are descriptions of the storage and process control hidden V-functions.

Type definitions

The definitions of the data types are shown in Figure 4. Only shown are those types that are more or less global in their use throughout the specification. Other types used in specific O-functions and V-functions will be presented as necessary.

Parameters and Global Definitions

Figure 5 lists all the parameters and quantified variables used in the specification, and Figure 6, some global definitions that apply to many of the function specifications. The parameters in Figure 5 are defined in terms of their type designations and need no further explanation. The meanings of the global definitions are somewhat more complex and are discussed below.

In most of the definitions shown in Figure 6 the symbol "Cur" is used. Cur is defined in each function in which it is used as a sub-

```
seg_type = integer (0 to max_seg_no);          /* segment number */

offset_type = structure                        /* offset into segment */
              (page: page_type,
               word_offset: word_offset_type);

page_type = integer (0 to max_pages);          /* page number in a segment */

word_offset_type = integer (0 to max_word_offset); /* no. words in page */

word_type = vector (integer (0 to word_length)) of boolean; /* machine word */

uid_type = vector (integer (0 to uid_length)) of boolean;   /* unique id */

pointer_type = structure                       /* virtual address pointer */
              (seg: seg_type,
               offset: offset_type);

classification_type = scalar ("unclassified", "confidential",
                      "secret", "top secret");

category_type = set (scalar ("nato", "nuclear", "china", ...));

level_type = structure
              (classification: classification_type,
               category: category_type);

access_level_type = structure
              (security_level: level_type,
               integrity_level: level_type);

user_id_type = vector (integer (0 to user_id_length)) of character;

type_type = scalar ("directory", "segment", "device"); /* object types */

entry_type = integer (0 to max_entries);       /* entry number of a branch */
```

Figure 4. Storage and Process Control Data Types

```
parameter

uid, uid1, uid2: uid_type;
dir: seg_type;
seg: seg_type;
entry: entry_type;
acl: acl_type;
access_level, al1, al2: access_level_type;
page: page_type;
mode: mode_type;
acl_mode: acl_mode_type;
offset: offset_type;
priority: priority_type;
proc: uid_type;
message: message_type;
destination_uid: uid_type;
source_uid: uid_type;
pointer: pointer_type;
entry_point: pointer_type;
word, compare_word: word_type;
initial_kst: set {seg_type};
pathname: pathname_type;
device_data: device_data_type;
physical_device: physical_device_type;
user_id: user_id_type;
SFEP: integer;

/* quantified variables */

ireg: reg_type;
ientry: entry_type;
idevice: device_type;
iacle: acle_type;
iuid: uid_type;
juid: uid_type;
i: integer;
iseg: seg_type;
ioffset: offset_type;
ipage: page_type;
```

Figure 5. Parameters

```
define
    dir_uid = Cur.kst(dir).uid;             /* uid of directory */
    dir_dir = Cur.kst(dir).dir;             /* segno of parent of dir */
    dir_entry = Cur.kst(dir).entry;         /* entry no. of dir */
    seg_uid = Cur.kst(seg).uid;             /* uid of segment */
    seg_dir = Cur.kst(seg).dir;             /* segno of parent of seg */
    seg_entry = Cur.kst(seg).entry;         /* entry no. of seg */
    Branch = Directory (dir_uid, entry); /* branch for [dir,entry] */
    Dir_branch = Directory (Cur.kst(dir_dir).uid, dir_entry);
                                            /* branch for dir */
    Seg_branch = Directory (Cur.kst(seg_dir).uid, seg_entry);
    device_dir = Cur.kdt(device).dir;       /* segno of parent of device */
    device_entry = Cur.kdt(device).entry; /* entry no. of device */
                                            /* branch for seg */
    Device_branch = Directory (Cur.kdt(device_dir), device_entry);
                                            /* branch for device */
```

Figure 6. Global Definitions

stitute for the reference to the V-function Process. For example, Cur may be defined as Process(uid), the value of Process for a given value of uid. In that case the value of dir_uid, the first definition in the figure, is Process(uid).kst(dir).uid, or the uid of the directory whose segment number is dir in the process uid. The definitions dir_dir and dir_entry are similarly the segment number of the parent and entry number of the specified directory, respectively. The three definitions seg_uid, seg_dir, and seg_entry are identical to the above three, except the argument is seg instead of dir.

Given the segment number of a directory ("dir") and the number of an entry ("entry") the value of Branch is the contents of the branch of the specified entry. Dir_branch and Seg_branch are respectively the branch for a directory and segment, given the segment number. The definitions device_dir, device_entry, and Device_branch have meanings similar to those for segments and directories.

Constants

The goal in the specification is to include as few literal constants as possible, since such constants may change as the design progresses and literal constants provide no information useful to the proof. Named constants are thus used wherever possible; the constants used in the specification are shown in Figure 7.

Process Hidden V-Function

Process is the main data base for each process that contains all process-dependent data. It is shown in Figure 8, along with the pertinent structures. It is indexed by unique-id of the process, and its components are as follows:

kst: This is the known segment table for each process, similar in function to the Multics KST [16]. It contains an entry for every initiated segment or directory. The segment number (called "seg" in the specification) of the object is used as an index into the kst. In a given process, there may not be more than one kst entry for any object. The following components are included in the kst:

type:	has the value "segment", "directory", or "device".
dir:	is the segment number of the parent directory.
entry:	is the entry number of the object in its parent directory.

```
constant

max_seg_no: integer;            /* maximum segment number */
max_pages: integer;             /* maximum number of pages in segment */
max_word_offset: integer;       /* maximum word offset in a page */
uid_length: integer;            /* maximum value of a uid */
user_id_length: integer;        /* number of characters in user_id */
max_entries: integer;           /* number of entries in a directory */
max_kst_size: integer;          /* maximum entries in kst */
max_acl_size: integer;          /* maximum number of entries in an acl */
word_length: integer;           /* length of machine word in bits */
max_message_length: integer;    /* length of data of message */
root_seg: seg_type;             /* segment number of root */
root_uid: uid_type;             /* uid of root */
root_parent_uid: uid_type;      /* uid of parent of root */
max_device_no: integer;         /* maximum device number */
max_pathname_length: integer;   /* maximum length of pathname */
interpreter_data_length: integer; /* size of interpreter data */
quit_flag: integer;             /* quit_flag index in interpreter data */
```

Figure 7. Constants

```
/* PROCESS - the process data structure */

Hidden_V_function Process(uid): process_type;

process_type = structure                         /* process data base */
                (kst: kst_type,
                 user_id: user_id_type,
                 uid: uid_type,
                 kdt: kdt_type,
                 priority: priority_type,
                 semaphore: integer,
                 messages: vector(uid_type) of message_type,
                 interpreter_data: interpreter_data_type,
                 access_level: access_level_type);


kst_type = vector (seg_type) of structure
                (type: type_type,
                 dir: seg_type,
                 entry: entry_type,
                 uid: uid_type,
                 wired_pages: vector (integer (0 to max_pages)) of boolean);


interpreter_data_type = vector (0 to inerpreter_data_length) of boolean;
```

Figure 8. Process V-Function

uid: is the unique-id of the object.

wired_pages: is an array of bits marking the pages of the segment that are currently "wired" by the current process. See the discussion of Wire and Unwire on page 78.

Each kst entry corresponds in some sense to a segment descriptor, and the entire kst corresponds to the descriptor segment.

user_id: This is the user-id associated with process.

kdt: The known device table is a structure in function similar to the kst. An initiated device has one entry in the kdt indexed by virtual device number. More detail on the structure of the kdt will be presented in Section VI.

priority: is a number, set by the user, specifying the execution priority of the process. The kernel places no restrictions on the value that may be set.

semaphore: is a semaphore indicating the number of interprocess communication messages currently stored in messages, below. Its value is incremented when a message is inserted and decremented when a message is removed.

messages: are the messages, indexed by unique-id of the message. The messages consist of arbitrary bit strings.

access_level: is the access level of the process.

interpreter_data: is unstructured data used by the interpreter for this process.

The value returned by Process is "undefined" for processes that do not exist. When a process is created, certain minimal information is inserted by Create_proc (see page 65) in order to allow the process to build its own kst, kdt, etc.

Directory Hidden V-Function

As stated previously, each object in the file system has a branch that contains the attributes of the object. A directory consists of a list of branches, one for each object in the directory. The V-function Directory, whose structure is shown in Figure 9, contains the contents of the branch for an object. Directory is indexed by unique-id of the directory and entry number of the branch. The contents of a

```
/* DIRECTORY - contents of a directory branch */

Hidden_V_function Directory (uid, entry): branch_type;


branch_type = structure                          /* branch in a directory */
               (uid: uid_type,
                type: type_type,
                access_level: access_level_type,
                acl: acl_type,
                device_data: device_data_type);


acl_type = structure                             /* acl of a branch */
               (size: acle_type,
                list: vector (acle_type) of structure
                    (user_id: user_id_type,
                     acl_mode: acl_mode_type));


mode_type = set              /* access modes that may be requested */
               (scalar ("read", "write", "execute",
                "status", "modify"));
```

Figure 9. Directory V-Function

branch, shown in the structure branch_type, are as follows:

uid: is the unique-id of the object.

type: is either "directory", "segment", or "device".

access_level: is the access level of the object.

acl: is the access control list (ACL) of the branch. The ACL of a branch is a vector of components containing a user-id and a set of modes. The modes that may be specified in a branch are any combination or none of "read", "write", and "execute". In order to determine whether a given type of access to an object is allowed, the modes in the ACL, the ring brackets, and the access level of the object must be examined.

device_data: If the object is a device, this information contains additional device attributes to be discussed in more detail in Section VI.

The kernel supports only those attributes of objects that are described above. Other attributes must be maintained by software.

Miscellaneous Hidden V-Functions and Macros

These last few V-functions are shown in Figure 10. Cur_proc simply holds the unique-id of the process currently running. Data, indexed by a segment's unique-id and offset within the segment, contains the words of data for a segment. Ancestor is a V-function that is set when an object is created. Given two unique-ids, Ancestor returns "true" if the first unique-id is that of an object which is a hierarchical ancestor of the second unique-id.

Unique_id is a V-function macro used throughout that returns a unique-id each time it is referenced. The unique-id is defined as an integer with some maximum value that will never be reached. In addition, each call to Unique_id must yield an integer that is greater than the previous value of Unique_id, but not otherwise dependent on it. The independence of consecutive values of Unique_id is imposed as a requirement in order to prevent modulating its value for use in transmitting information. The requirement that Unique_id be constantly increasing allows the value of an object's unique-id to be conveniently used to determine the object's relative age (see the discussion of Wakeup on page 67). In an implementation, the most obvious choice for the value of Unique_id would be the real-time clock value as indicated in the specification.

```
/* CUR_PROC - unique-id for current process */

Hidden_V_function Cur_proc: uid_type;


/* DATA - contents of data segments */

Hidden_V_function Data (uid, offset): word_type;


/* ANCESTOR - "true" if uid1 is an ancestor of uid2 */

Hidden_V_function Ancestor (uid1, uid2): boolean;


/* UNIQUE_ID - returns a unique number constantly increasing */

V_function_macro Unique_id: uid_type;

derivation
    "current time in microseconds since 0000 GMT 1 January 1901"
end;
```

Figure 10. Miscellaneous Hidden V-functions

INITIATION AND TERMINATION

An entry (directory, segment or device) in a directory is identified by an entry number within the directory. Before an entry can be accessed in any way, directly by machine instructions or indirectly with kernel calls, it must be "initiated". The act of initiation of a directory or segment results in the assignment of a segment number to the entry. Initiation of a device yields a device number. The device number and segment number are virtual addresses chosen by uncertified software and saved by the kernel. After initiation, most references to the object may be made by segment or device number. For initiated segments and devices, these references are made by uncertified software using hardware machine instructions, and address translation and access is controlled by hardware. References to initiated directories are made interpretively using kernel calls. Initiation of segments and directories will be discussed below. Device initiation will be discussed in detail in Section VI.

In order to initiate an object, the parent directory must first be initiated. The Initiate O-function shown in Figure 11 requires as input parameters the segment number of the parent directory, the entry number of the object, and the segment number to be assigned to the object. Initiation is allowed if the security checks are satisfied and if there is not already an object initiated with that segment number.

Note that only the non-discretionary security checks are made at initiate time. The non-discretionary checks can be made because the information on which these checks are based (i.e., the access levels of the object and process) will not change while the object is initiated.[7] Discretionary security checks, however, involve data (the access control lists) that can be changed at the top level after the object is initiated. Thus, discretionary checks must be made at each access to the object -- not just at initiate time. A check of discretionary security by Initiate would be needless.

Terminate, shown in Figure 12, simply sets the specified kst entry undefined. The important considerations in Terminate are the exceptions. The second exception checks that a directory cannot be terminated if entries in that directory are still initiated. This exception guarantees that all initiated objects have an initiated parent, thereby assuring that the branches for all initiated objects are ac-

---

[7]Upgrading of an object is accomplished simply by a process at the upgraded access level copying the object. Downgrading must be done external to the system, e.g., by writing a tape at one access level and then reading it at a lower access level. Trusted functions can also be given the ability to downgrade.

```
/* INITIATE - initiates a segment or directory */

O_function Initiate (dir, entry, seg);

let  Cur = Process(Cur_proc);

exception
     Cur.kst(dir) = "undefined";
     Cur.kst(seg) ≠ "undefined";
     Cur.kst(dir).type ≠ "directory";
     Branch = "undefined";
     ^Dominates (Cur.access_level, Branch.access_level);
     Branch.type = "device";

effect
     Cur.kst(seg).dir = dir;
     Cur.kst(seg).entry = entry;
     Cur.kst(seg).uid = Branch.uid;
     Cur.kst(seg).type = Branch.type;
     Cur.kst(seg).wired_pages = "false";
end;
```

Figure 11. Initiate O-function

```
/* TERMINATE - terminates a segment or directory */

O_function Terminate (seg);

let  Cur = Process(Cur_proc);

exception
    Cur.kst(seg) = "undefined";
    Cur.kst(seg).type = "directory" &
        ((∃iseg)(Cur.kst(iseg).dir = seg) |
         (∃idevice)(Cur.kdt(idevice).dir = seg));
    Cur.kst(seg).type = "segment" &
        (∃ipage)(Cur.kst(seg).wired_pages(ipage) = "true");

effect
    Cur.kst(seg) = "undefined";
end;
```

Figure 12. Terminate O-Function

cessible. The third exception checks that a segment to be terminated does not have any pages that are currently wired. Wiring of pages will be discussed under Input/Output in Section VI.

ACCESS CONTROL

In Figure 13 four V-function macros are shown that are used by other top level functions to make access checks relating to discretionary or non-discretionary security. These macros are typically referenced on every access that requires the check.

Inas ("in address space") is called on every access to a segment or directory. Input parameters are the unique-id of the process on behalf of which the access is requested, the segment number, the access mode desired ("read", "write", "execute", "status", or "modify"). The result is either "true" or "false". Inas uses the access level and the access control list of the object. A check is also made to see if the object is still initiated, and if the desired access mode is appropriate for the entry type (e.g., "status" can only be requested for a directory).

Access_permission is a macro that scans the ACL of an initiated segment or directory, and returns "true" or "false" if the user_id of the current process is on the list with the desired access mode. Note that the first occurrence of an entry that matches the user_id is used to find the mode. The kernel makes no check of what is put on the ACL (See Set_acl on page 53) and therefore it is possible for more than one entry to specify the same user_id.

Dominates is used throughout the specification to compare two access levels. The value returned by Dominates is "true" if the first access level dominates the second. This V-function macro is the only place in the specification that cares about the structure of the access level and how two access levels are compared. Thus, it would be a simple matter to change the access level structure (e.g., to remove integrity).

In Figure 14 are shown two primitives for setting and reading access control lists. Setting an ACL of an object requires modify permission to the parent directory, and reading the ACL requires status permission. Since the object itself is not referenced by these primitives the object need not be initiated or accessible. It must, however, exist. No check at all is made for validity of the contents of the ACL -- that is entirely the responsibility of the user or supervisor.

```
/* INAS - determines if process has access to object */

V_function_macro Inas (proc, seg, mode): boolean;

let  Cur = Process(proc);
     may_read = Access_permission (proc, seg, "read");
     may_write = Seg_branch.access_level = Cur.access_level &
                        Access_permission (proc, seg, "write");
     may_execute = Access_permission (proc, seg, "execute");

derivation
     Inas = (Cur.kst(seg) ≠ "undefined") &
            (caseof mode
               "read"!    (Seg_branch.type = "segment") & may_read;
               "write"!   (Seg_branch.type = "segment") & may_write;
               "execute"! (Seg_branch.type = "segment") & may_execute;
               "status"!  (Seg_branch.type = "directory") & may_read;
               "modify"!  (Seg_branch.type = "directory") & may_write;)
end;


/* ACCESS_PERMISSION - "true" if process has requested access on ACL */

V_function_macro Access_permission (proc, seg, acl_mode): boolean;

let  Cur = Process(proc);
     acle = min {iacle | iacle ≤ Seg_branch.acl.size &
                 Seg_branch.acl.list(iacle).user_id = Cur.user_id};

derivation
     Access_permission = (∃iacle)(iacle ≤ Seg_branch.acl.size &
               Seg_branch.acl(iacle).user_id = Cur.user_id &
               acl_mode ∈ Seg_branch.acl(acle).mode);
end;
```

Figure 13. Access Checking V-Function Macros

```
/* DOMINATES - returns "true" if first access level dominates second */

V_function_macro Dominates (al1, al2): boolean;

let  sl1 = al1.security_level;
     sl2 = al2.security_level;
     il1 = al1.integrity_level;
     il2 = al2.integrity_level;

derivation
     Dominates = ((sl1.category = sl2.category &
                     sl1.classification ≥ sl2.classification) |
                   (sl2.category ⊆ sl1.category &
                     sl1.classification = sl2.classification)) &
                 ((il1.category = il2.category &
                     il1.classification ≤ il2.classification) |
                   (il1.category ⊆ il2.category &
                     il1.classification = il2.classification));
end;
```

Figure 13. Access Checking V-Function Macros (concluded)

```
/* GET_ACL - obtains the acl of an object */

V_function Get_acl (dir, entry): acl_type;

let  Cur = Process(Cur_proc);

exception
     ^Inas (Cur_proc, dir, "status");
     Branch = "undefined";

derivation
     Get_acl = Branch.acl;
end;


/* SET_ACL - sets the acl of an object */

O_function Set_acl (dir, entry, acl);

let  Cur = Process(Cur_proc);

exception
     ^Inas (Cur_proc, dir, "modify");
     Branch = "undefined";

effect
     Branch.acl = acl;
end;
```

Figure 14. ACL Primitives

CREATION AND DELETION

Creation of objects in the hierarchy is allowed for segments and directories, and deletion is allowed for all objects. Devices cannot be created by the user or supervisor because devices correspond to physical objects that exist externally and whose attributes are externally determined. The structure of the Input/Output subsystem and device objects will be discussed in detail in Section VI. Primitives for creating devices are discussed in Section VII as trusted functions.

Create is shown in Figure 15 and Delete is shown in Figure 16. Both creation and deletion require modify access to the parent directory, thus the access level of the directory must be equal to that of the process. The arguments to Create and Delete include the segment number of the directory and the entry number of the branch. For Create, the caller chooses the entry number for the object to be created and the access level of the object to be created.

An object may be created having any access level greater than or equal to the current access level of the process. The Create primitive provides the only means by which an object of a greater access level can be created. Once such an upgraded segment or directory is created, it cannot be initiated for access by the current process. The branch of the upgraded object, however, can be accessed.

In addition to initializing the branch, Create sets the value of the V-function Ancestor to indicate that the parent directory is an ancestor of the newly created entry. Ancestor is also set to indicate that all ancestors of the parent directory are now ancestors of the new entry.

Though no attributes, other than the access level, are specified when using Create, the kernel must initialize attributes in the branch so that they will be defined in case of subsequent initiation of the object. Since creation of an object does not necessarily imply that it will be subsequently initiated by the same process, the ACL is set to null. Thus, if the object is to be used after creation, its ACL must first be set for access by the current process. Also, the data for the contents of segments is zeroed.

Delete allows unrestricted deletion of any object (segment, directory, or device) as long as the process has "modify" permission to the parent directory. Note that although the user is not allowed to create a device, there is no security violation if the user can delete the device. Deletion of a segment or device is accomplished simply by calling the O-function macro Delete_object, which deletes the branch for the object (see discussion of Delete_object below). It is not

```
/* CREATE - creates a directory or segment */

O_function Create (dir, entry, type, access_level);

let  Cur = Process(Cur_proc);

exception
     type = "device";
     ^Inas (Cur_proc, dir, "modify");
     ^Dominates (access_level, Dir_branch.access_level);
     Branch ≠ "undefined";

effect
     Branch.uid = ´Unique_id;
     Branch.type = type;
     Branch.access_level = access_level;
     Branch.acl.size = 0;
     if type = "segment" then
          (∀ioffset)(Data(Branch.uid,ioffset)=0);
          end;
     Ancestor(dir_uid, Branch.uid) = "true";
     (∀iuid)(if Ancestor(iuid, dir_uid)
             then Ancestor(iuid, Branch.uid) = "true";
             end);
end;
```

Figure 15. Creating an Object

```
/* DELETE - deletes a segment, directory, or device */

O_function Delete (dir, entry);

let  Cur = Process(Cur_proc);

exception
     ^Inas (Cur_proc, dir, "modify");
     Branch = "undefined";

effect
     if ´Branch.type = "directory"
     then (∀iuid)(if Ancestor(´Branch.uid, iuid) = "true"
                  then effects_of Delete_object (iuid);
                  end);
     end;
     effects_of Delete_object (´Branch.uid);
end;


/* DELETE_OBJECT - deletes the branch of an object */

O_function_macro Delete_object (uid);

effect
     (∀iuid,ientry)
          (if ´Directory(iuid,ientry).uid = uid
           then if ´Directory(iuid,ientry).type = "device"
                then (∀juid,idevice)
                          (if ´Process(juid).kdt(idevice).uid = uid
                           then Process(juid).kdt(idevice) = "undefined";
                           end);
                else (∀juid,iseg)
                          (if ´Process(juid).kst(iseg).uid = uid
                           then Process(juid).kst(iseg) = "undefined";
                           end);
                end;
                Directory(iuid,ientry) = "undefined";
           end);
end;
```

Figure 16. Deleting Objects

necessary to explicitly delete the Data for the segment since, once the uid of the segment in the branch is deleted, that data can no longer be referenced.

The deletion of a directory is more complex because it involves the possible deletion of many objects inferior to that directory. The Multics kernel does not allow deletion of non-empty directories due to the complexity involved and the fact that maintaining quota information is impossible when upgraded directories can be deleted by a process of a lower access level. For the SFEP kernel, deletion of an upgraded directory, and all its inferiors, is merely a "write up", which is permissible. Delete uses Ancestor to find all descendants of a directory to be deleted, and Delete_object is called to delete each of the descendants.

Note that it would be considerably simpler to not allow for deletion of non-empty directories, as in Multics. However, if deletion of an upgraded directory is to be allowed, the user cannot be told (via an exception) whether that upgraded directory is empty. Thus in order to allow for deletion of upgraded directories, a facility must be provided to delete the entire subtree.

There is an alternative approach that allows for deletion of upgraded subtrees without requiring the kernel to scan for descendants. This approach requires the user to create upgraded processes of the proper levels, and passing them the unique-ids of the upgraded directories whose contents are to be deleted. The kernel need only then provide a function that deletes a single object, given a unique-id. The problem is that, although the Delete primitive is simplified, inconsistencies in the hierarchy possible in case of user error or malfunction can result in objects that have no parent directory, and thus cannot be referenced. These problems have implications in other places in the specification where it is assumed that all initiated objects in the kst have an existing parent. The additional check required to handle these "lost objects", combined with the requirement for a facility that periodically purges such objects, was seen to be no less complicated than the full implementation of a Delete_object primitive.

Delete_object actually deletes the branch of an object, thereby making its contents (the value of Directory or Data) inaccessible. Whenever an object is deleted, the kernel must also remove all kst and kdt entries for all processes that may have initiated the object. This makes it impossible to access the object by segment or device number.

MANIPULATION OF ATTRIBUTES

Two V-functions are provided that return attributes of objects, Get_attributes and Get_entries shown in Figure 17. For reasons discussed previously, these primitives are not required for security but are required to provide a utility. There are no additional primitives for setting attributes other than the ACL primitives in Figure 14, because the remaining attributes cannot be changed after the object is created.

Get_attributes returns a structure containing all of the kernel maintained information stored in the branch, except for the unique-id, which is hidden, and the ACL, which is provided by Get_acl in Figure 14. The object may be of any type. In addition to providing utility, Get_attributes is useful to observe file system changes made by the SSO.

Get_entries, which returns a set of entry numbers in a directory that are in use, is in the same class as Get_attributes in that the supervisor could theoretically maintain this information itself. Abstractly, though, Get_entries is not even required for utility, since it would be possible for the supervisor to infer the set of entry numbers by using Get_attributes with all possible entry numbers and noting which ones returned exceptions. This function is provided, however, to allow the supervisor to determine the structure of the hierarchy in a "clean" and efficient manner.

HARDWARE PRIMITIVES

The "interpreter concept" discussed in Section II provides the rationale for putting certain hardware-implemented functions in the top level specification, and for leaving others out. The four basic hardware primitives are shown in Figure 18. All four functions take a pointer as an argument and are used to access a location in memory. Read and Execute are V-functions that return the contents of the location referenced, and Write is an O-function that writes a word of data into a segment. Test_and_set is an OV-function that both reads and writes a word conditionally, and is used to provide an indivisible read-alter-rewrite operation necessary for implementing locks.

Although the interpreter is capable of reading an instruction using the Read primitive, the discretionary access policy requires that a primitive that checks for "execute" access be provided. Thus, it is assumed that the interpreter will use Execute for instruction fetches, and Read for data fetches.

```
/* GET_ATTRIBUTES - obtains type and access level of an object */

V_function Get_attributes (dir, entry): structure
                                        (type: type_type,
                                         access_level: access_level_type);

let  Cur = Process(Cur_proc);

exception
     ^Inas (Cur_proc, dir, "status");
     Branch = "undefined";

derivation
     Get_attributes.type = Branch.type;
     Get_attributes.access_level = Branch.access_level;
end;


/* GET_ENTRIES - returns a set of all valid entries in a directory */

V_function Get_entries (dir): {entry_type};

let  Cur = Process(Cur_proc);

exception
     ^Inas (Cur_proc, dir, "status");

derivation
     Get_entries = {ientry | Directory(dir_uid, ientry) ≠ "undefined"};
end;
```

Figure 17. Reading Attributes

```
/* READ - reads a word from a segment into a register */

V_function Read (pointer): word_type;

let  Cur = Process(Cur_proc);

exception
     ^Inas (Cur_proc, pointer.seg, "read");

derivation
     Read = Data (Cur.kst(pointer.seg).uid, pointer.offset);
end;


/* WRITE - writes a word of data in a segment */

O_function Write (pointer, word);

let  Cur = Process(Cur_proc);

exception
     ^Inas (Cur_proc, pointer.seg, "write");

effect
     Data (Cur.kst(pointer.seg).uid, pointer.offset) = word;
end;


/* EXECUTE - read a word from a segment using execute permission */

V_function Execute (pointer): word_type;

let  Cur = Process(Cur_proc);

exception
     ^Inas (Cur_proc, pointer.seg, "execute");

effect
     Execute = Data (Cur.kst(pointer.seg).uid, pointer.offset);
end;
```

Figure 18. Hardware Primitives

```
/* TEST_AND_SET - reads and writes a word conditionally */

OV_function Test_and_set (pointer, compare_word, word): word_type;

let  Cur = Process (Cur_proc);
     seg_word = Data (Cur.kst(pointer.seg).uid, pointer.offset);

exception
     ^(Inas (Cur_proc, pointer.seg, "write") &
       Inas (Cur_proc, pointer.seg, "read"));

effect
     if 'seg_word = compare_word then seg_word = word; end;

derivation
     Test_and_set = 'seg_word;
end;
```

Figure 18. Hardware Primitives (concluded)

Read_interpreter_data and Write_interpreter_data, shown in Figure 19, provide access to the interpreter data for the process. These two primitives, plus the four primitives discussed above for referencing memory, can be combined to provide for the execution of all hardware instructions and associated operations such as faults and interrupts.

```
/* READ_INTERPRETER_DATA - read all interpreter data */

V_function Read_interpreter_data: interpreter_data_type;

derivation
     Read_interpreter_data = Process(Cur_proc).interpreter_data;
end;


/* WRITE_INTERPRETER_DATA - writes all of interpreter data */

O_function Write_interpreter_data (interpreter_data);

effect
     Process(Cur_proc).interpreter_data = interpreter_data;
end;
```

Figure 19. Reading and Writing Interpreter Data

SECTION V

PROCESS CONTROL

PROCESS CREATION AND DELETION

Create_proc and Delete_proc are shown in Figure 20. Creation of a process with a specified user_id at an equal or higher access level is allowed. When a process is created, it is given a list of segment numbers of segments to be pre-initiated in the new process's kst, called an initial_kst. Create_proc must verify that the new kst to be created is consistent -- that parent directories of all entries specified are also specified. The new Process data base is filled in as required, and the kst of the new process is initialized. If the new process is of a higher access level than the original process, the new process will not have write access to any segments in its address space, and will thus not be able to do much until it initiates or creates a segment at its level. The interpreter_data of the new process, however, is available for "scratch" storage until writable segments are initiated. The Create_proc primitive provides almost the simplest possible mechanism that will create a process with a given address space and allow it to begin execution. Note that little complexity is added to the specification by allowing a set of segments to be specified by the original process, instead of just one segment (at least one segment is needed).

Delete_proc will delete any process with an access level equal to or greater than that of the current process. No check of user_id is made, since deletion of a process with a different user_id is a denial of service issue.

It is intended that the Create_proc and Delete_proc be used to implement the system function that creates and destroys processes for users as they log in and out. The operating system most likely will not allow arbitrary user processes to create or delete other processes with different user_ids. A single process, usually referred to as the root process or answering service, will be responsible for creating and deleting user processes as users log in and out.

The ability to create another process with a different user_id may appear to be a violation of discretionary security, since it would be possible for a process, which does not have access to a particular segment, to create another process as its agent with the proper user_id to access the segment. The discretionary access rules, however, do not require a validated user authentication mechanism to be within the kernel. They only require that the access control mecha-

```
/* CREATE_PROC - Create another process of equal or higher access level */

OV_function Create_proc (user_id, initial_kst, access_level): uid_type;

let  Cur = Process(Cur_proc);
     New = Process('Unique_id);

exception
     ^Dominates (access_level, Cur.access_level);
     ^(∀iseg)(if iseg ∈ initial_kst then
          iseg = root_seg |
          Cur.kst(iseg) ≠ "undefined" & Cur.kst(iseg).dir ∈ initial_kst;
          end);

effect
     New.uid = 'Unique_id;
     New.user_id = user_id;
     New.kdt = "undefined";
     New.priority = Cur.priority;
     New.access_level = access_level;
     (∀iseg)(if iseg ∈ initial_kst
             then New.kst(iseg) = Cur.kst(iseg);
             else New.kst(iseg) = "undefined";
             end);

derivation
     Create_proc = New.uid;
end;


/* DELETE_PROC - Delete another process */

O_function Delete_proc (uid);

let  Cur = Process(Cur_proc);

exception
     Process(uid) = "undefined" |
                    Process(uid).acess_level ≠ Cur.access_level;

effect
     Process(uid) = "undefined";
end;
```

Figure 20. Creating and Deleting Processes

nism itself be within the kernel. Since it not possible to verify that ACLs are properly set according to the (human) user´s wishes, it is of little use to try to verify that user authentication, upon which the ACLs are based, works properly. Of course, since user authentication can be administratively controlled, and placed into the supervisor or special processes such as the answering service, there is no problem implementing a meaningful authentication mechanism (i.e., assignment of user_id) outside the kernel that is protected from user tampering.

In order to allow for an answering service running outside the kernel to be capable of creating a process of any access level, Create_proc and Delete_proc allow reference to a process of a higher access level. Thus, the answering service would run at a system low access level. When the signal to create a new process is received due to a "login" signal from some device, (see the discussion of Connect on page 97), the answering service must create a segment in some known place (having a specific pathname or a specific segment number) that contains the device´s pathname, and then call Create_proc, specifying the proper access level and user_id. If further user authentication is to take place, by reading more input from the device, this authentication must be done by a process that can initiate the device -- not the answering service that is at system low. The new process may be a system initializer process that performs this authentication and finally creates the user process with the proper user_id. A subsequent logout or disconnect, which is again signalled to the answering service, must be followed by a wakeup to the system initializer process, telling it to destroy the user process. The initializer process (or the answering service) can then destroy itself. Note that, because it is not possible to obtain the unique-id of a process created by another process of a higher access level, the answering service can only send a wakeup to the initializer process and not the user process.

The previous paragraph only explains one method by which processes on the SFEP can come into being. It is a sufficient example to illustrate that the Create_proc and Delete_proc primitives provide the necessary functionality.

INTERPROCESS COMMUNICATION

Interprocess communication involves the coordination and signalling of events between processes. The kernel provides primitives for process coordination for two reasons. First, since the kernel is responsible for process scheduling (even though that scheduling is transparent), a facility must be provided that allows a process to schedule itself and to wait for events. Second, interprocess communication allows processes of various access levels to send messages to a single process of a higher access level. Using regular data segments to send such messages results in performance penalties such as the possibility of lost messages or messages that can never be deleted.

The Process V-function contains two items used in interprocess communication: a queue of messages and a semaphore. The semaphore is simply a count of messages in the queue. The message queue is a vector indexed by unique-id of a message that contains arbitrary data inserted by the sending process. Also, if the message is from an I/O device (see Device_interrupt in Section VI) the message identifies the source.

The functions used for interprocess communication and scheduling are illustrated in 21. The Process data structure and message queue structure are also illustrated. Set_priority is a feature seen necessary in a communications processor for process scheduling. It is assumed that the kernel somehow uses this value to schedule the current process appropriately. Unfortunately, the specification of this O-function yields effects that are only visible to the outside world in terms of real time. It is not clear what the implication is of an O-function whose effects are not visible at the top level.

Block is the O-function that allows a process to go into a blocked or "wait" state until a certain event occurs. This event is defined to be the appearance of a message in the queue. When a message is in the queue, the semaphore is decremented and the message is returned as a value. If there are no messages in the queue when Block is called (i.e., the semaphore is initially zero), the process waits. The first two lines of the effect, stating that the semaphore must be positive and must be decremented by one, cannot both be satisfied unless the semaphore is first incremented by some other process. The specification of Block is somewhat non-standard in the sense that a wait is implied, though no wait is actually visible to the current process.

Note that the message removed by Block is the one whose unique-id has the smallest value. Since the definition of Unique_id, discussed on page 45, states that the value is always increasing, the message with the lowest value is also the oldest.

Wakeup, together with the O-function macro Send_wakeup, inserts a message into the queue of another process and increments the semaphore. The only exception signalled by Wakeup is when the process's access level is less than that of the current process. Wakeup must not allow the caller to distinguish between a nonexistent process and a process of a higher access level. If the process to which the message is sent does not exist, the message is simply not sent and the caller is not notified. In order to actually implement sending of a

```
process_type = structure                           /* process data base */
              (kst: kst_type,
               user_id: user_id_type,
               uid: uid_type,
               kdt: kdt_type,
               priority: priority_type,
               semaphore: integer,
               messages: vector(uid_type) of message_type,
               interpreter_data: interpreter_data_type,
               access_level: access_level_type);


message_type = vector (message_length_type) of boolean;


message_length_type = integer (0 to max_message_length);


priority_type = integer;


/* SET_PRIORITY - Set the execution priority of the process */

O_function Set_priority (priority);

let  Cur = Process(Cur_proc);

effect
     Cur.priority = priority;
end;
```

Figure 21. Interprocess Communication Primitives

```
/* BLOCK */

OV_function Block: message_type;

let  Cur = Process(Cur_proc);
     uid = min {iuid | ´Cur.messages(iuid) ≠ "undefined"};

effect
     Cur.semaphore ≥ 0;
     Cur.semaphore = ´Cur.semaphore - 1;
     Cur.messages(uid) = "undefined";

derivation
     Block = ´Cur.messages(uid);
end;


/* WAKEUP */

O_function Wakeup (proc, message);

let  Cur = Process(Cur_proc);

exception
     Process(proc) ≠ "undefined" &
       ^Dominates (Process(proc).access_level, Cur.access_level);

effect
     if Process(proc) ≠ "undefined"
     then Process(proc).semaphore = Process(proc).semaphore + 1;
          Process(proc).messages(´Unique_id) = message;
     end;
end;


/* SEND_WAKEUP - sends a message to a process */

O_function_macro Send_wakeup (proc, message);

effect
     Process(proc).semaphore = Process(proc).semaphore + 1;
     Process(proc).messages(´Unique_id) = message;
end;
```

Figure 21. Interprocess Communication Primitives (continued)

```
/* INTERROGATE - examine next message without going blocked */

OV_function Interrogate: message_type;

let  uid = min {iuid | 'Cur.messages(iuid) ≠ "undefined"};
     Cur = Process(Cur_proc);

exception
     (∀ iuid) (Cur.messages(iuid) = "undefined");

effect
     Cur.semaphore = Cur.semaphore - 1;
     Cur.messages(uid) = "undefined";

derivation
     Interrogate = 'Cur.messages(uid);
end;
```

Figure 21. Interprocess Communication Primitives (concluded)

message to a process of a higher access level, it is necessary that the unique-id of the receiving process be somehow communicated to the process at the lower access level. This can only happen if the receiving process was created by a process of an equal or lower access level than the sending process, or if such information is provided externally.

The last function, Interrogate, allows a message to be removed from the queue without the process's going blocked if the queue is empty.

SECTION VI

INPUT/OUTPUT

DEVICES IN HIERARCHY

I/O devices are objects in the hierarchy that have several attributes in addition to those for segments and directories. The structure of a directory branch, repeated in Figure 22, contains an item called device_data. When a device is created using a trusted function, the device is given a minimum and maximum access level, both of which must be greater than or equal to the access level of the parent directory. The "current" access level of the device, valid when the device is "connected", is the actual access level of the object as used by kernel primitives accessed by non-kernel software. The minimum and maximum levels are administratively determined and are physical parameters that have to do with the physical environment in which the device is located. When a user, using a trusted function, connects a device, he specifies an access level that must be between the minimum and maximum for the device, and that access level becomes the current access level of the object.

HARDWARE SUPPORT OF I/O

The structure of the I/O subsystem supported by the kernel is heavily based on the SFEP hardware support of I/O. The virtual memory and address translation mechanism have already been discussed as they relate to storage control. I/O devices are also supported in the virtual memory. Figure 23 illustrates the address translation performed by the SFEP to convert a virtual device number into a physical device reference. Note the similarity between this figure and Figure 3 for memory accesses. The DBR, in the top of the figure, is used to point to the base of the set of descriptors for I/O devices, and the virtual device number is an index into this table. The I/O descriptor contains the physical device number that hardware uses to access the device.

The reason that there is only one level of descriptor for device references is that there is no need to split the descriptor segment into pages (since the number of I/O descriptors is usually small) and there is no reason to break up the virtual device address. In addition, since the kernel does not allow devices to be shared, there is no need to allow for sharing of device descriptors.

```
branch_type = structure                         /* branch in a directory */
              (uid: uid_type,
               type: type_type,
               access_level: access_level_type,
               acl: acl_type,
               device_data: device_data_type);


device_data_type = structure                    /* branch info for devices */
              (physical_device: physical_device_type,
               min_access_level: access_level_type,
               max_access_level: access_level_type,
               connected: boolean);


physical_device_type = integer;                 /* physical device address */
```

Figure 22. Device Attributes

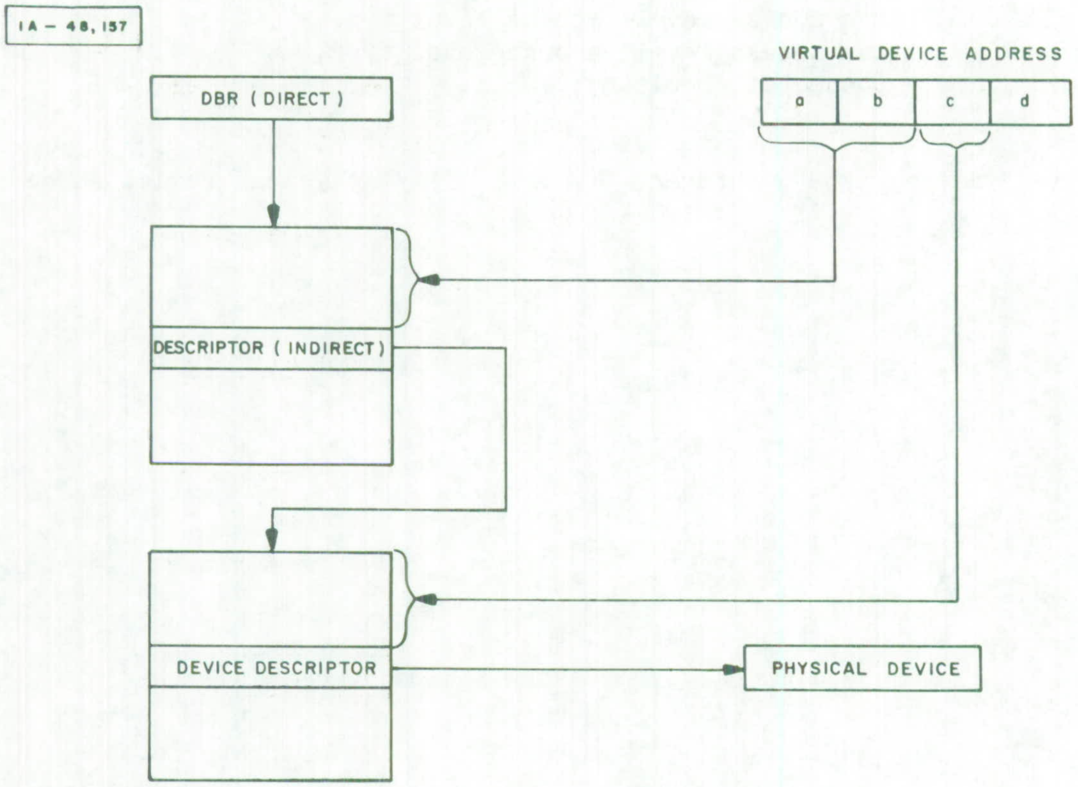


Figure 23. DEVICE ADDRESS TRANSLATION

Figure 23 only illustrates the controls involved when a process accesses a device directly. Also to be considered is a device's access to memory, in the case where the device is a direct memory access (DMA) device. The SFEP hardware supports DMA devices securely by simply mapping all memory references made by such devices in exactly the same manner as processor memory references are mapped. The situation is somewhat complicated by the fact that a DMA device may request access to memory when the process originating the I/O request is not currently running, but hardware takes care of this problem by keeping a correspondence between DBR values and physical devices having I/O currently in progress.

There is also an alternative implementation of DMA memory references that the SFEP supports. Note that, since a memory reference may involve several levels of descriptor references, hence many accesses to memory, it may not be possible to support high speed DMA devices with such a structure. Even at relatively low speeds, performance is certainly degraded somewhat when every access to memory requires up to three additional fetches. The performance problem is alleviated to a certain degree by using associative memories for descriptors, but in order to provide for full performance capability the SFEP hardware also allows a "premapped" mode of I/O.

In premapped mode the necessary access checks to memory are made at time of initiation of the I/O operation ("initiation" of an I/O operation must not be confused with initiation of the device in the kdt to be discussed below). All subsequent memory references made by a premapped device are made to absolute addresses. The disadvantages of premapped I/O are in certifiability. In order to certify that premapped I/O is secure, it is necessary to first determine unambiguously all pages of memory that might be accessed in a given I/O operation, verify that access, make sure the pages will be "in core" throughout the duration of the I/O, and finally "trust" the I/O controller not to reference any memory outside of its assigned area. It is not believed that it is possible to certify this type of I/O, and thus premapped I/O will be ignored in the top level kernel specification. Of course, at lower levels within the kernel, where the invoking software is correct and trusted, it may be very useful to provide premapped I/O for support of virtual memory on disks or tapes.

INITIATION AND TERMINATION

The Initiate_device O-function, along with the kdt structure, is shown in Figure 24. The caller of Initiate_device provides the directory number and entry number of the device and the virtual device number he wants to use. The rules for device initiation are exactly the same as those for segment and directory initiation, except that the

```
device_type = integer (0 to max_device_no);  /* virtual device address */


kdt_type = vector (device_type) of structure /* known device table */
                (dir: seg_type,
                 entry: entry_type,
                 uid: uid_type,
                 device_reg: word_type,
                 physical_device: physical_device_type);


/* INITIATE_DEVICE - initiates a device */

O_function Initiate_device (dir, entry, device);

let  Cur = Process(Cur_proc);

exception
     Cur.kdt(dir) = "undefined";
     Cur.kdt(device) ≠ "undefined";
     Cur.kdt(dir).type ≠ "directory";
     Branch = "undefined";
     Cur.access_level ≠ Branch.access_level;
     Branch.type ≠ "device";
     Branch.device_data.connected = "false";
     (∃ iuid, idevice)
                 (Process(iuid).kdt(idevice).device_data.physical_device =
                  Branch.device_data.physical_device);

effect
     Cur.kdt(device).dir = dir;
     Cur.kdt(device).entry = entry;
     Cur.kdt(device).uid = Branch.uid;
     Cur.kdt(device).device_data.physical_device =
                       Branch.device_data.physical_device;
end;
```

Figure 24. Device Initiation

access level of the device must equal that of the process (because devices are always initiated for read-write), and there is an additional check (the last exception in the figure) that no other process has currently initiated the device. The kernel cannot allow more than one process to simultaneously start I/O on the same device, because when the device later makes a reference to memory, it is impossible to determine on behalf of which process the reference is being made. Since the starting of I/O on a device is performed by software without kernel intervention, the kernel has no way of remembering which process last accessed the device. Only by limiting the initiation of a device to one process can the device´s access be subsequently monitored.[8]

The known device table (kdt) data structure requires some explanation. The components of the kdt are listed below, along with their meanings.

dir: segment number of the parent directory for this device.

entry: entry number of the device in its parent directory.

uid: unique-id of the device. Note that if physical device numbers are unique, it would be possible to use the physical device number instead of the uid. The uid is used, however, to maintain consistency in the file system that associates each object with a uid. Moreover, when a device object is deleted (by a trusted function), it is necessary that the unique-id not be reused when the same physical device is recreated.

device_reg: This is an abstract device register which is loaded or stored by user software. The values loaded into this register can be thought of as corresponding to the various opera-

---

[8]Since it is hardware itself that monitors the starting of an I/O operation and the subsequent memory references made by the device, it may be argued that hardware could safely keep track of which process last referenced the device. Such monitoring, however, is difficult to implement because it involves too many interactions with device controllers (e.g., determining whether the previous operation has finished, whether the device is ready to accept a new operation, to whom to return status information, etc.). In reality, the device controller itself will probably refuse to accept an I/O operation until the previous operation has completed, but controllers cannot be relied upon to provide this form of protection. Unfortunately, to save on hardware costs there are generally such things as multiple device controllers that control more than one device. See the discussion of asynchronous devices near the middle of page 85.

tions that the device may perform, and values read from this register can be considered to be status information. This register can also be used to read and write data to the device if the device supports synchronous I/O.

physical_device: is the physical device number.

Terminate_device, illustrated in Figure 25, needs to make none of the checks required by Terminate, except to signal an exception if the device is already terminated or does not exist.

WIRE AND UNWIRE

When a DMA device makes references to memory, it cannot afford to get a page fault, because most DMA devices cannot recover from faults as software can. Since page swapping is not visible to non-kernel software, and hence not controllable by users, the only way to guarantee that a page will be "in core" or "wired" when a device attempts to access that page is to provide a primitive that instructs the kernel to keep the page wired. It is the responsibility of the supervisor or user to guarantee that all pages accessed by I/O devices are wired. In order for the kernel to completely hide page swapping, it is necessary that a trap or fault be signalled whenever a device, acting on behalf of a given process, attempts to reference a page not wired by that process.

Unfortunately, it is possible for an unwired page to be in core at the right time, thereby resulting in a successful access by a device, and the SPM hardware does not provide a facility for forcing device faults on unwired pages without also forcing faults on software´s access to those pages. Such a situation involving paging activity can be used to transmit information: a top secret process, by the rate at which it accesses an unclassified page, can control whether that page is in core or not. If the "page in core" condition were visible to some other unclassified process, an information channel would exist. The "page in core" condition is normally not visible, because the kernel hides paging for memory references. For I/O accesses to memory, though, the success or failure of an I/O operation reveals clearly whether a page is in core. If the I/O operation is initiated by the unclassified process, the "page in core" condition is therefore visible. The ability of the top secret process to directly control paging can be eliminated by not allowing the explicit wiring of a page of a lower access level. The implicit control of paging, however, (i.e., that caused by memory references by a process) cannot be prevented.

This information channel presented by paging activity may be too "noisy" or of too low a bandwidth to be used successfully. In order

```
/* TERMINATE_DEVICE - terminates a device */

O_function Terminate_device (device);

let  Cur = Process(Cur_proc);

exception
    Cur.kdt(device) = "undefined";

effect
    Cur.kdt(device) = "undefined";
end;
```

Figure 25. Terminating a Device

to keep the specification itself secure, the primitives that are used by a device to read and write memory are specified as if an exception is always signalled if the page has not been wired by the associated process. (See the Device_read_memory V-function in Figure 28.) Because pages can only be wired if they are at a level equal to that of the process, it therefore also becomes impossible to perform I/O on a page of a different access level.

Wire and Unwire are shown in Figure 26. Note that Wire only allows wiring of a page of a segment with an access level equal to that of the current process. Unwire needs no such check, since a page of a different access level could not have been wired in the first place.

ACCESS TO AND BY DEVICES

I/O devices generally come in two forms, asynchronous and synchronous. Synchronous I/O usually involves a transfer of data between a device and a hardware register, and thus can be considered to be similar to control operations or operations that initiate I/O by a device. Asynchronous I/O, in which the device itself references memory without the help of the process, must be treated as if the process that is using the device is making the references.

Synchronous access to an I/O device is provided by the V-function Read_device_register and O-function Write_device_register shown in Figure 27. The exceptions checked by these functions are similar to those checked by Inas for access to segments. No equivalent Inas for devices is provided, since only these two functions require the type of test made by Inas.

Access to memory by a device is provided by the Asynchronous functions Device_read_memory and Device_write_memory illustrated in Figure 28. Because these are asynchronous functions, they can be called at any time in any process. Thus, before making any accesses, the process for which the I/O is being performed must be determined. The first two "lets" in the specification of each function obtain the uid of the process that last initiated the device. The exceptions check that there is indeed some process that has currently initiated the device (to take care of the possibility that a DMA device is terminated or deleted before finishing its operation), that access to the memory location by the process is allowed in the specified mode, and that the page in which the word appears is wired by the process. See the discussion of Wire and Unwire on page 78 for the motivation behind this latter check.

When a device has completed an I/O operation, it normally signals an interrupt to the processor. The kernel must then route this inter-

```
/* WIRE - wires a page in real memory */

O function Wire (seg, page);

let  Cur = Process(Cur_proc);

exception
     Cur.kst(seg) = "undefined";
     Cur.kst(seg).type ≠ "segment";
     Seg branch.access level ≠ Cur.access_level;

effect
     Cur.kst(seg).wired pages(page) = "true";
end;


/* UNWIRE - unwires a page in memory */

O function Unwire (seg, page);

let  Cur = Process(Cur_proc);

exception
     Cur.kst(seg) = "undefined";
     Cur.kst(seg).type ≠ "segment";

effect
     Cur.kst(seg).wired_pages(page) = "false";
end;
```

Figure 26. Wiring Pages

```
/* READ_DEVICE_REGISTER - reads I/O device register */

V_function Read_device_register (device): word_type;

let  Cur = Process(Cur_proc);
     acle = min {iacle | Device_branch.acl(iacle).user_id = Cur.user_id};

exception
     Cur.kdt(device) = "undefined";
     ^(∃iacle)(Device branch.acl(iacle).user_id = Cur.user_id &
          "read" ∈ Device_branch.acl(acle).mode);

derivation
     Access_device_register = Cur.kdt(device).device_reg;
end;


/* WRITE_DEVICE_REGISTER - loads device register, possibly starting I/O */

O function Write_device_register (device, word);

let  Cur = Process(Cur_proc);
     acle = min {iacle | Device_branch.acl(iacle).user_id = Cur.user_id};

exception
     Cur.kdt(device) = "undefined";
     (∃iacle)(Device_branch.acl(iacle).user_id = Cur.user_id &
          "write" ∈ Device_branch.acl(acle).mode);

effect
     Cur.kdt(device).device_reg = word;
end;
```

Figure 27. Reading and Writing Device Registers

```
/* DEVICE_READ_MEMORY - device read memory */

Asynchronous_V_function Device_read_memory
                     (physical_device, pointer): word_type;

let  (proc, device) = (iuid, idevice) ∍
          (Process(iuid).kdt(idevice).device_data.physical_device =
                    physical_device);
     Cur = Process(proc);

exception
     ^(∃iuid, idevice)
          (Process(iuid).kdt(idevice).device_data.physical_device =
                    physical_device);
     ^Inas (proc, pointer.seg, "read");
     ^Cur.kst(pointer.seg).wired pages(pointer.offset.page);

derivation
     Device_read_memory = Data(Cur.kst(pointer.seg).uid, pointer.offset);
end;


/* DEVICE_WRITE_MEMORY - device write into memory */

Asynchronous_O_function Device_write_memory
                     (physical_device, pointer, word);

let  (proc, device) = (iuid, idevice) ∍
          (Process(iuid).kdt(idevice).device_data.physical_device =
                    physical_device);
     Cur = Process(proc);

exception
     ^(∃iuid, idevice)
          (Process(iuid).kdt(idevice).device_data.physical_device =
                    physical_device);
     ^Inas (proc, pointer.seg, "write");
     ^Cur.kst(pointer.seg).wired_pages(pointer.offset.page);

effect
     Data(Cur.kst(pointer.seg).uid, pointer.offset) = word;
end;
```

Figure 28. Device Access to Memory

```
/* DEVICE_INTERRUPT - Interrupt a process by a device */

Asynchronous_O_function Device_interrupt (physical_device, message);

effect
     (∀ iuid, idevice)
         (if Process(iuid).kdt(idevice).device_data.physical_device =
                physical_device
          then effects of Send wakeup (iuid, bit(idevice)||message);
          end;);
end;
```

Figure 29. Device Interrupt

rupt to the proper process that is currently handling the device. Interrupts as such are not supported by the kernel. Instead, interrupts from devices (or timers) are transformed into wakeups, and the process in charge of the device must examine its wakeup messages. In Figure 29, the Asynchronous Device_interrupt O-function is shown. This primitive is called by the device when it wishes to signal an interrupt. The parameters are the physical device number and an arbitrary message that becomes the data associated with the interrupt. The interrupt then simply becomes a wakeup to the process currently using the device. The virtual device number, identifying the source of the interrupt, must also be made available to the process, and, since the device itself cannot supply the virtual device number as part of the message, the kernel inserts it.

Note that the secure operation of these asynchronous functions is dependent upon the correctness of the "physical_device" parameter that identifies the device requesting the access. In the proof of the specification it must be stated, as an assertion, that this physical_device parameter is the identifier of the same device that was addressed when the I/O operation was initiated. For many devices, the kernel can either determine the physical device number by virtue of the channel number or port to which the device is physically connected, or must trust the device to correctly identify itself. For multiple device controllers, where one single channel may be used for several devices, the kernel must further trust the controller to provide the proper identification of the device.

In the case of single channel devices, there is a certain amount of risk in trusting a device to identify itself, unless the device controller is certified to provide the correct information. This certification is not seen as too difficult of a problem, because the hardware providing the identification of the device is probably relatively simple. For multiple device controllers, the hardware that distinguishes between devices is usually much more complex, and may involve a microprogram. If such controllers cannot be certified, they must be restricted to single level operation (i.e., all devices on a given controller must have the same access level). Although the SFEP kernel treats all devices as independent, the ability to assign minimum and maximum access levels to each device allows for restricted operation of groups of devices on a single controller.[9]

[9]Identification of the device is not the only security related problem associated with multiple line controllers. The fact that such controllers must correctly and securely handle information of various access levels simultaneously presents an even greater verification problem.

HOST-SFEP COMMUNICATION

The main function of the SFEP is, of course, to act as a front-end processor for the host Multics computer. In the top level specification, very few additional functions need be provided to support communication with the host. Figure 30 illustrates the primitives used in communication with Multics. In the specification of these two functions in the figure, the SFEP must reference host kernel primitives. To avoid ambiguity with SFEP primitives of the same name, the prefix "Host_" is inserted in front of any reference to a host function.

Messages are sent between SFEP and the host via the interprocess wakeup mechanism. Communication between two processes in this manner is restricted to processes of the same access level. To send a message to the host, the SFEP process calls Send in Figure 30, and supplies the unique-id of the host process and the message to be sent. The Host_Send_wakeup O-function macro, used to send the message, operates in a manner very similar to the SFEP Send_wakeup illustrated in Figure 21.

The host has a primitive, just like Send, that has as its effects the SFEP Send_wakeup. Thus, receipt of a message from the host can be accomplished by the SFEP by using the standard interprocess communication facilities Block and Interrogate.

Because there is a necessity for the SFEP to signal some kind of interrupt in the host process, for handling the Multics QUIT signal, a predetermined bit of the interpreter data of the host, called the "quit_flag", that can be directly set by an SFEP process, is used to signal an interrupt. The host interpreter presumably examines the quit_flag at appropriate intervals. The Interrupt O-function is called by the SFEP process to set the quit_flag in a host process´s interpreter data.

There is no required capability for a host process to interrupt an SFEP process. This is because, since the SFEP´s major task is I/O, and not general purpose programming, the SFEP process can be counted on to call Block or Interrogate often enough to see even the most urgent messages coming from the host.

NETWORK COMMUNICATION

The Multics SFEP must also be able to communicate to other network processors in its role as a secure communications processor. Single level networks present no problem for non-kernel software. Multiple level secure networks, however, must be handled by the kernel

```
/* SEND - send a message to host process */

O_function Send (proc, message);

let  Cur = Process(Cur_proc);

exception
     Host_Process(proc) ≠ "undefined" &
       ^Dominates (Host_Process(proc).access_level, Cur.access_level);

effect
     effects of Host_Send_wakeup (proc, message);
end;


/* INTERRUPT - Send interrupt to host process */

O_function Interrupt (proc);

let  Cur = Process(Cur_proc);

exception
     Host_Process(proc) ≠ "undefined" &
       ^Dominates (Host_Process(proc).access_level, Cur.access_level);

effect
     Host_Interpreter_data(proc) [quit_flag] = "true";
end;
```

Figure 30. Host-SFEP Communication Primitives

and therefore primitives must be provided for communication through such a network. In the SFEP specification, it is assumed that all multilevel network interfaces are to other SFEPs. An interface to an arbitrary multilevel network cannot be specified because of the arbitrary nature of network protocols. A "generic" type of specification that may adequately deal with several types of protocols existing in presently available secure networks is apt to be too abstract to be of much use, or too complex to verify.

The O-function SFEP_send, shown in Figure 31, is used by an SFEP process to send a message to a process in another SFEP. The function is almost identical to Send for sending a message to the host. The only difference is the checking of a value, treated here as a constant bit vector, called SFEP_exists, to determine whether there is an SFEP of the specified name. This vector could also have been specified as a V-function with a trusted O-function to set it, but such an additional complication is, in an abstract sense, unnecessary.

```
SFEP_exists: vector (integer) of boolean;


/* SFEP_SEND - Send message to another SFEP process */

O_function SFEP_send (SFEP, proc, message);

let  Cur = Process(Cur_proc);

exception
     ^SFEP_exists(SFEP);
     SFEP_Process(proc) ≠ "undefined" &
          ^Dominates (SFEP_Process(proc).access_level, Cur.access_level);

effect
     effects of SFEP_Send_wakeup (SFEP, proc, message);
end;
```

Figure 31. Network Communication

SECTION VII

TRUSTED FUNCTIONS

A detailed explanation of trusted functions and their relationship to the top level specification has been discussed in Section II on page 20. In the SFEP, trusted functions are utilized to provide certain functions that cannot be implemented at the top level due to possible *-property violations. This section discusses a preliminary set of trusted functions in terms of O-functions (and supporting V-functions) that a trusted user or System Security Officer (SSO) calls directly from his console. Since the specification is abstract, these O-function calls are specified just like the top level O-function calls. It is the understanding, however, that the arguments passed to these O-functions are read directly from the user's terminal, without processing by any non-kernel software. Code conversion and terminal protocols are implicitly included in the specification of a trusted function, and may not be handled by software outside the trusted function interface. Of course, in an actual implementation there will have to be software to read input from the terminal and process that input before passing it on as a function call. As far as the verification is concerned, all software and hardware between the user's console and the kernel are part of the trusted function.

It was stated previously that incorrect invocation of trusted functions could possibly result in a security compromise in terms of a *-property violation. No amount of checking, via exceptions, can prevent this. It would therefore seem unnecessary to provide for any exception conditions in the specification of these functions, because exceptions do not help to verify that the function is correctly invoked.

It must be remembered, however, that the "proof" of a trusted function is a proof of correctness, not a proof of security. An exception in a trusted function is not placed there to make security checks, but is used because the exception is part of the desired functionality to be implemented. Since trusted functions are invoked by people, and people (even trusted ones) can make mistakes, part of the operation of a trusted function may be to help catch inadvertent mistakes. For example, certain errors in the invocation of a trusted function could result in an inconsistency in the hierarchy (such as the creation of a segment without a parent directory) that might be difficult to detect if not checked at the time the trusted function is invoked. Exceptions appearing in a trusted function then are simply considered to be part of the primitive's operation.

V-FUNCTION MACROS

These V-function macros provide values to the kernel that the trusted user has no particular need to see. Each of the macros shown in Figure 32 provides derived values useful in the implementation of other O-functions and V-functions used by trusted functions.

Since trusted functions are invoked by users and not by processes, there is no concept of a kst or of a current process. Therefore, in order to allow a trusted user to access an object in the hierarchy, a means must be provided by which an object can be named, without using segment numbers. One possibility is the use of unique-id´s, since the V-functions Directory and Data describe all objects in terms of their unique-id´s. Unique-id´s, however, are difficult for users to enter and are prone to human error. In addition, since an object´s attributes are in the parent, a user who knows only the unique-id of an object has no simple way of finding the attributes. Therefore, the concept of a "pathname" of an object is recognized. A pathname, whose structure is defined by pathname_type, shown at the top of Figure 32 on page 92, is composed of a series of entry numbers and a length. The last number in the pathname is the entry number of the object in its parent directory. The number before that is the entry number of the parent directory in its parent, and so on. The first entry number is that of the first directory under the root. A pathname having zero length identifies the root itself.

Since a pathname by itself is not useful to the primitives in finding an object, most of the macros are involved in the conversion of a pathname to a uid and vice versa. Parent_path, the first macro in Figure 32, returns the pathname of the parent of the object. This pathname is simply the same as the argument minus its last entry.

Path_to_uid returns the unique-id of the object described by a pathname. The unique-id of the root is a system constant, root_uid. Note that the derivation of this macro is recursive, making use of Parent_path to find the pathname and the unique-id of the parent. Although it has been the policy in the specification to avoid recursion in O-functions, due to the difficulty seen in verification, there seems to be no problem with recursive V-function macros. Uid_to_path is the inverse macro that returns the pathname for a given unique-id.

Path_exists, similarly recursive, returns "true" if the specified object exists. This macro is used in exceptions to verify a given pathname. Finally, Parent_uid returns the unique-id of the parent directory of an object.

```
pathname_type = structure                    /* pathname of an object */
              (length: pathname_length_type,
               entries: vector (pathname_length_type) of entry_type);


pathname_length_type = integer (0 to max_pathname_length);


/* PARENT_PATH - Returns pathname of parent */

V_function_macro Parent_path (pathname): pathname_type;

derivation
     if pathname.length = 0
     then Parent_path = pathname;
     else (∀i)(if 1≤i≤pathname.length-1
               then Parent_path.entries(i) = pathname.entries(i);
               end);
          Parent_path.length = pathname.length - 1;
     end;
end;


/* PATH_TO_UID - Returns unique-id of pathname */

V_function_macro Path_to_uid (pathname): uid_type;

derivation
     if pathname.length = 0
     then Path_to_uid = root_uid;
     else Path_to_uid = Directory (Path to_uid (Parent_path(pathname)),
                                   pathname.entries(pathname.length)).uid;
     end;
end;
```

Figure 32. Trusted Subjects V-Functions

```
/* UID_TO_PATH - Returns pathname for a uid */

V_function_macro Uid_to_path (uid): pathname_type;

let  parent_pathname = Uid_to_path (Parent_uid (uid));

derivation
     if uid = root_uid
     then Uid_to_path.length = 0;
     else
          (∀i)(if 1≤i≤parent_pathname.length
               then Uid_to_path.entries(i) = parent_pathname.entries(i);
               end);
          Uid_to_path.entries(parent_pathname.length+1) = ientry ∍
                    Directory (Parent_uid(uid), ientry).uid = uid;
          Uid_to_path.length = parent_pathname.length + 1;
     end;
end;


/* PATH_EXISTS - Returns "true" if branch for object exists */

V_function_macro Path_exists (pathname): boolean;

derivation
     if pathname.length = 0
     then Path_exists = "true";
     else Path_exists = Path_exists (Parent_path (pathname)) &
             Directory(Path_to_uid(Parent_path(pathname)),
                       pathname.entries(pathname.length)) ≠ "undefined";
     end;
end;


/* PARENT_UID - Returns uid of parent, given a uid */

V_function_macro Parent_uid (uid): uid_type;

derivation
     Parent_uid = iuid ∍
                    (∃ientry)(Directory(iuid,ientry).uid = uid);
end;
```

Figure 32. Trusted Subjects V-Functions (concluded)

DEVICE MANIPULATION

Devices are the one object type that cannot be created by user software. The reason is that in order to create a device in the hierarchy the physical device number must be supplied and minimum and maximum access levels must be specified, information about which non-kernel software has no access or knowledge. Also, since an exception must be signalled if that same physical device already exists somewhere in the hierarchy, even if it is at a higher access level, a possible information channel would exist.

The Create_device O-function is shown in Figure 33. Input parameters are the pathname of the directory in which the device is to be created, the entry number of the device, and the device_data structure containing the physical device number and the minimum and maximum access levels. The first exception checks that the device does not already exist in the hierarchy and the second checks that the pathname is valid. The additional exceptions check that the consistency relation

$$\text{parent access level} \leq \text{minimum access level} \leq \text{maximum access level}$$

where "$\leq$" means "dominates", holds true. Finally, the "connected" component of device_data is checked to insure that it is initially off.

Note that if Create_device were available to untrusted software at the top level kernel interface, a clear security violation would exist in the first exception, unless the caller was a system_high process. If the caller is system_high, then a *-property violation would exist when the device is created. There is no way to provide for secure creation of devices by non-kernel software unless devices are removed from the hierarchy or placed in a restricted place.

The same reasoning behind the motivation for providing V-functions such as Get_acl and Get_attributes leads to the need for a V-function that returns the pathname of a device in the hierarchy. The V-function Device_path, in Figure 34, returns a physical device's pathname if it exists.

Note that other operations on devices, such as deletion and setting of the ACL, can be performed by untrusted software using primitives already defined at the top level.

```
/* CREATE_DEVICE - Creates a branch for a physical device */

O_function Create_device (pathname, entry, device_data);

let  parent_uid = Path_to_uid (pathname);

exception
     (∃iuid,ientry) (Directory(iuid,ientry).type = "device" &
           Directory(iuid,ientry).device_data.physical device =
                device_data.physical_device);
     ^Path_exists (pathname);
     Directory (parent_uid, entry) ≠ "undefined";
     ^Dominates (device_data.min_access_level,
        Directory (Path_to_uid (Parent_path (pathname)),
                pathname.entries(pathname.length)).access_level);
     ^Dominates (device_data.max_access_level, device_data.min_access_level);
     device_data.connected = "true";

effect
     Directory (parent_uid, entry).uid = ´Unique_id;
     Directory (parent_uid, entry).type = "device";
     Directory (parent_uid, entry).access_level = device_data.min_access_level;
     Directory (parent_uid, entry).acl.size = 0;
     Directory (parent_uid, entry).device_data = device_data;
end;
```

Figure 33. Creating a Device

```
/* DEVICE_PATH - Returns pathname of a physical device */

V_Function Device_path (device_data.physical_device): pathname_type;

let  device_exists = Directory(iuid,ientry).type = "device" &
           Directory(iuid,ientry).device_data.physical_device =
                                        device_data.physical_device;

exception
     ^(∃iuid,ientry)(device_exists);

derivation
     (∀iuid,ientry)
          (if device_exists
           then Device_path = Uid_to_path (Directory(iuid,ientry).uid);
           end);
end;
```

Figure 34. Obtaining the Pathname of a Device

LOGGING IN AND OUT

The SFEP kernel must support a trusted interface that logs in a terminal at a given access level. This access level is selected by the user when the terminal is first "connected" to the system, in some manner such as by typing in a value or pressing a special key. Once the terminal is assigned an access level by the kernel, non-kernel software can initiate the device as it pleases.

Figure 35 shows the Connect and Disconnect functions that are invoked by the user, in a trusted mode, to log a terminal in or out. The first parameter to Connect, the physical device number, is not supplied by the user but is available by virtue of the physical connection. The access level is entered by the user and, if the device exists in the hierarchy and the selected access level is between the minimum and maximum allowed for the device, the device is defined to be "connected". Once connected, any process of the level of the device may initiate it. In order to provide for the signalling of a connection to software, the kernel sends a wakeup to a particular process defined by non_kernel software to be the "answering service". The hidden primitive V-function Answering_service_uid is set by software using Set_answering_service_uid to the uid of the answering service process, for use in the Connect operation. The wakeup message itself consists of the pathname of the device that has been connected.

Disconnect occurs via some signal from the terminal, and thus the physical device number is the only parameter required. The kernel terminates the kdt entry for the device in the process that initiated it, and resets the connected flag.

The physical realization of the connect and disconnect procedure must reasonably reflect the operation of these two functions. In the simplest case, the power on and power off interrupts, received by the kernel, can be intercepted and interpreted as connect and disconnect functions.

```
/* ANSWERING_SERVICE_UID - uid of answering service used for logging in */

Hidden_V_function Answering_service_uid: uid_type;


/* SET_ANSWERING_SERVICE_UID - set unique-id of answering service */

O_function Set_answering_service_uid (uid);

effect
     Answering_service_uid = uid;
end;


/* CONNECT - login a device */

O_function Connect (physical_device, access_level);

let  device_exists = Directory(iuid,ientry).type = "device" &
       Directory(iuid,ientry).device_data.physical_device = physical_device;
    (dir_uid, entry) = (iuid, ientry) ≻ (device_exists);

exception
     ^(∃ iuid, ientry)(device_exists);
     ^Dominates (access_level, Branch.min_access_level) |
          ^Dominates (Branch.max_access_level, access_level);
     Branch.connected = "true";

effect
     Branch.connected = "true";
     Branch.access_level = access_level;
     effects_of Send_wakeup (Answering_service_uid,
                                 bit (Device_path (physical_device)));
end;
```

Figure 35. Logging In and Out

```
/* DISCONNECT - logout physical device */

O_function Disconnect (physical_device);

let device_exists = Directory (iuid, ientry).type = "device" &
       Directory(iuid,ientry).device_data.physical_device = physical_device);
    (dir_uid, entry) = (iuid, ientry) ∋ (device_exists);

exception
     ^(∃iuid,ientry) (device_exists);

effect
     (∀ iuid, idevice)
         (if ´Process(iuid).kdt(idevice).physical_device = physical_device
          then Process(iuid).kdt(idevice) = "undefined";
               end);
     Branch.connected = "false";
end;
```

Figure 35. Logging In and Out (concluded)

SECTION VIII

CONCLUSION

The security kernel specified in this report provides a general purpose interface that allows for the use of the kernel not only in an SFEP, but in general applications including that of a secure communications processor. The design of the kernel draws heavily from experience with the Multics kernel, being very similar in the file system and virtual memory organization. The SFEP kernel is intended to run on a specific minicomputer currently being designed by Honeywell to provide a hardware base for a secure front-end processor and communications processor. When verified and certified, the SFEP and Multics kernels will provide for the implementation of a large scale secure multi-level operating system.

APPENDIX I

SPECIFICATION SYNTAX

The syntax is expressed in extended BNF. The characters "+" and "*" are interpreted to mean "zero or more" and "one or more" of the preceeding element, respectively. Punctuation consisting of semicolons and commas has been omitted from the syntax specification, but may be freely used in the specification for clarity. Braces {} are used to surround required sections consisting of multiple symbols, and brackets [] surround optional sections. Comments in the specification surrounded by /* and */ are ignored. Underlining is used to indicate terminal symbols that might otherwise be confused with symbols of the meta-language. The terminal character strings of the language are <number> (any string of numeric characters) and <symbol> (any string of characters, not all numeric).

```
<module>::=
        module <module_name>;
        <type>*
        <define>*
        <parameter>*
        <constants>*
        <V_function>*
        <Hidden_V_function>*
        <OV_function>*
        <O_function>*
        <Asynchronous_O_function>*
        <Asynchronous_V_function>*
        <O_function_macro>*
        <V_function_macro>*

<module_name>::=
        <symbol>

<type>::=
        type {<type_name>+ = <type_designator>;}+

<type_name>::=
        <symbol>

<type_designator>::=
        <simple_type>
        | <constructed_type>
```

```
<simple_type>::=
        <scalar_type>
        ¦ <subrange_type>
        ¦ <type_name>

<scalar_type>::=
        scalar(<constant>+)

<constant>::=
        "<symbol>"
        ¦ <number>

<subrange_type>::=
        <simple_type>(<constant> to <constant>)

<constructed_type>::=
        <vector_type>
        ¦ <structure_type>
        ¦ <set_type>

<vector_type>::=
        vector <subrange_type> of <simple_type>

<structure_type>::=
        structure({<field_name>+ : <simple_type>}+)

<field_name>::=
        <symbol>

<set_type>::=
        set(<simple_type>)

<define>::=
        define {<identifier> = <expression>;}+

<identifier>::=
        <symbol>

<parameter>::=
        parameter {<fp_name>+ : <type_designator>;}+

<fp_name>::=
        <symbol>

<constants>::=
        constant {<identifier>+ : <type_designator>[ = <expression>];}+
<expression>::=
        <constant>
        | <identifier>
        | <function_call>
        | ´<function_call>
        | <any reasonable mathematical operation on expressions>

<function_call>::=
        <fn_name>[(<expression>+)]

<fn_name>::=
        <symbol>

<V_function>::=
        V_function <fn_name>[(<fp_name>+)] : <type_designator>
        [let <let>+]
        [exception <exception>+]
        [derivation <value>]
        end

<let>::=
        <identifier> = <expression>;

<exception>::=
        <expression>;

<value>::=
        <expression>;

<Hidden_V_function>::=
        Hidden_V_function <fn_name>[(<fp_name>+)] : <type_designator>
        [[let <let>+]
        [derivation <value>]
        end]

<OV_function>::=
        OV_function <fn_name>[(<fp_name>+)] : <type_designator>
        [let <let>+]
        [exception <exception>+]
        [effect <effect>+]
        [derivation <value>]
        end

<effect>::=
        <expression>;

<O_function>::=
        O_function <fn_name>[(<fp_name>+)]
        [let <let>+]
        [exception <exception>+]
        [effect <effect>+]
        end

<Asynchronous_V_function>::=
        Asynchronous_V_function <fn_name>[(<fp_name>+)] : <type_designator>
        [let <let>+]
        [exception <exception>+]
        [derivation <value>]
        end

<Asynchronous_O_function>::=
        Asynchronous_O_function <fn_name>[(<fp_name>+)]
        [let <let>+]
        [exception <exception>+]
        [effect <effect>+]
        end

<O_function_macro>::=
        O_function_macro <fn_name>[(<fp_name>+)]
        [let <let>+]
        [effect <effect>+]
        end

<V_function_macro>::=
        V_function_macro <fn_name>[(<fp_name>+)] : <type_designator>
        [let <let>+]
        derivation <value>
        end
```

APPENDIX II

INDEX TO FUNCTIONS

Below is a list of all function and type names used in the specification, and the figure and page on which they are illustrated.

REFERENCES

1. J.P. Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, Volume I and II, James P. Anderson & Co., Fort Washington, Pennsylvania, October 1972.

2. E.I. Organick, The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Massachusetts, 1972.

3. Multics Programmers' Manual, AG91 revision 1, AG92C revision 1, AG93C revision 1, and AK92 revision 1, Honeywell Information Systems Inc., July 1976.

4. W.L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45," ESD-TR-75-69, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, May 1975 (AD A011712).

5. Honeywell Information Systems, "Design For Multics Security Enhancements," ESD-TR-74-176, Electronic Systems Division (AFSC), Hanscom AF Base, Massachusetts, December 1973.

6. L. Smith, "Architectures for Secure Computer Systems," ESD-TR-75-51, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, April 1975 (AD A009221).

7. Honeywell Information Systems, Level 6 Minicomputer Handbook, AS22, January 1976.

8. D.E. Bell and L.J. LaPadula, "Secure Computer Systems," ESD-TR-73-278, Volumes I-III, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, November 1973-June 1974.

9. D.E. Bell and L.J. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation," ESD-TR-75-306, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, March 1976 (AD A023588).

10. K.J. Biba, "Integrity Considerations for Secure Computer Systems," ESD-TR-76-372, Electronic Systems Division, Hanscom AF Base, Massachusetts, April 1977 (AD A039324).

11. M. D. Schroeder and J. H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," _Communications of the ACM_, Volume 15, Number 3, March 1972, pp. 157-170.

12. D. L. Parnas, "A Technique for Software Module Specification with Examples," _Communications of the ACM_, Volume 15, Number 5, May 1972, pp. 330-336.

13. W. R. Price, "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1973.

14. N. Wirth, "The Programming Language PASCAL," _Acta Informatica_, Volume 1, 1971, pp. 35-63.

15. Honeywell Information Systems, _The Multics Virtual Memory_, AG95, June 1972.

16. A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory: Concepts and Design," _Communications of the ACM_, Volume 15, Number 5, May 1972, pp. 308-318.